SOFTWARE DEVELOPMENT SERIES

MANX™

Aztec C68k
for the Amiga

version 3.2
April 1986

Copyright (c) 1986 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

Distributed by:
Manx Software Systems, Inc.
P.O. Box 55
Shrewsbury, N.J. 07701
201-542-2121

USE RESTRICTIONS

The components of the Aztec C68k software development system are licensed software products. Manx Software Systems reserves all distribution rights to these products. Use of these products is prohibited without a valid license agreement. The license agreement is provided with each package. Before using any of these products the license agreement must be signed and mailed to:

Manx Software Systems
P. O. Box 55
Shrewsbury, N. J 07701

The license agreement limits use of these products to one machine. Any uses of these products that might lead to the creation of or distribution of unauthorized copies of these products will be a breach of the licensing agreement and Manx Software Systems will excercise its right to reclaim the original and any and all copies derived in whole or in part from first or later generations and to pursue any appropriate legal actions.

Software that is developed with Aztec C68k software development system can be run on machines that are not licensed for these products as long as no part of the Aztec C software, libraries, supporting files, or documentation is distributed with or required by the software. In the latter case a licensed copy of the appropriate Aztec C software is required for each machine utilizing the software. There is no licensing required for executable modules that include runtime library routines.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at 52.227-7013. DAC #84-1, 1 March 1984. DOD Far Supplement.

COPYRIGHT

Copyright (C) 1981, 1982, 1984 by Manx Software Systems. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without prior written permission of Manx Software Systems, Box 55, Shrewsbury, N. J. 07701.

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Manx Software Systems to notify any person of such revision or changes.

TRADEMARKS

Aztec C68k, Manx AS, Manx LN, and Z are trademarks of Manx Software Systems. Amiga is a trademark of Commodore-Amiga, Inc. CP/M-86 is a tradmark of Digital Research. MSDOS is a trademark of Microsoft. PCDOS is a trademark of IBM. UNIX is a trademark of Bell Laboratories. Macintosh is a trademark of Apple Computer.

Manual Revision History

February 1986 ..... First Edition
April 1986 ..... Second Edition

Summary of Contents

Amiga-specific Chapters

title	*code*
Overview	ov
Tutorial Introduction	tut
The Compiler	cc
The Assembler	as
The Linker	ln
Utility Programs	util
Library Functions Overview: Amiga Information	libov68
Aztec C68k/Amiga Functions	lib68
Amiga Functions	libamiga
Technical Information	tech
Debugging Utilities	debug

System Independent Chapters

Overview of Library Functions	libov
System-Independent Functions	lib
Style	style
Compiler Error Messages	err

Contents

Overview ..... ov

Tutorial Introduction ..... tut
1. Getting Started 3
   1.1 Copying the disks 3
   1.2 Disk usage during development 4
2. Using the CLI 4
   2.1 Looking around 4
   2.2 Working with files 6
3. Compiling, assembling, and linking 7
4. Environment variables 8
5. The ram disk 8
6. Where to go from here 9

The compiler cc
1. Operating Instructions 3
   1.1 The C Source File 3
   1.2 The Output Files 3
   1.3 #*include* files 5
   1.4 Memory Models 7
2. Compiler Options 13
   2.1 Summary of Options 13
   2.2 Description of Options 15
3. Programmer Information 19
   3.1 Supported Language Features 19
   3.2 Structure Assignment 19
   3.3 Structure Passing 19
   3.4 Line Continuation 19
   3.5 The *void* Data Type 19
   3.6 Special Symbols 20
   3.7 String Merging 20
   3.8 Long Names 21
   3.9 Reserved Words 21
   3.10 Global Variables 21
   3.11 Data Formats 21
   3.12 Register Usage 22
   3.13 In-line Assembly Language Code 23

3.14 Writing Machine-Independent Code 23
4. Error Processing 26

The Assembler ..... as
1. Operating Instructions 3
1.1 The Input File 3
1.2 The Object Code File 4
1.3 Listing File 4
1.4 Optimizations 4
1.5 Searching for *include* Files 4
2. Assembler Options 6
3. Programmer information 8

The Linker ..... ln
1. Introduction to linking 3
2. Using the Linker 7
3. Linker Options 9

Utility Programs ..... util
acvt 4
adump 6
arcv 7
cmp 8
cnm 9
diff 13
grep 17
hd 23
lb 24
make 35
mkarcv 7
obd 52
ord 53
set 54
setdate 55
Z 56

Library Overview: Amiga Information libov68

Amiga Functions lib68
Index 4
The functions 5
Amiga Functions ..... libamiga

Technical Information ..... tech
Program Organization 4
Code Segmentation 5
Libraries 8
Interfacing to Assembly Language 9
Interrupt Handlers 12

Debugging Utilities ..... debug
db (program debugger) 4
1. Overview 5
1.1 Basic Commands 5
1.2 Names 5
1.2.1 Code and Data Symbols 6
1.2.2 Operator Usage of Names 6
1.3 Loading programs and symbols 6
1.4 Breakpoints 7
1.5 Memory-change breakpoints 8
1.6 Trace mode 8
1.7 Backtracing 8
1.8 Macros 9
1.9 Displaying source files 9
1.10 Other features 9
2. Using DB 10
2.1 Starting DB 10
2.2 Using DB with an external terminal 10
2.3 Commands 11
2.3.1 Definitions 11
2.4 Command descriptions 15
2.4.1 The BREAKPOINT (b) commands 15
2.4.2 The CLEAR (c) commands 18
2.4.3 The DISPLAY (d) commands 18
2.4.4 The Exit (e) command 21
2.4.5 The 'Find source string' (f) command 22
2.4.6 The GO (g) commands 22
2.4.7 The LOAD (l) commands 23
2.4.8 The MODIFY MEMORY (m) commands 25
2.4.9 The Radix (n) command 26
2.4.10 The PRINT (p) command 27
2.4.11 The QUIT (q) command 32
2.4.12 The REGISTER (r) command 33
2.4.13 The SINGLE STEP (s/t) commands 33
2.4.14 The UNASSEMBLE (u) commands 34
2.4.15 The VARIABLE (v) commands 34
2.4.16 The MACRO (x) command 35
2.4.17 The Swap Screen (') command 35
2.4.18 The 'Display Expression' command 35
2.4.19 The HELP (?) command 36
3. Command Summary 37

Overview of Library Functions ..... libov
1. I/O Overview 4
1.1 Pre-opened devices, command line args 4
1.2 File I/O 6
1.2.1 Sequential I/O 6
1.2.2 Random I/O 6

1.2.3 Opening Files 6
1.3 Device I/O 7
1.3.1 Console I/O 7
1.3.2 I/O to Other Devices 7
1.4 Mixing unbuffered and standard I/O calls 7
2. Standard I/O Overview 9
2.1 Opening files and devices 9
2.2 Closing Streams 9
2.3 Sequential I/O 10
2.4 Random I/O 10
2.5 Buffering 10
2.6 Errors 11
2.7 The standard I/O functions 12
3. Unbuffered I/O Overview 14
3.1 File I/O 15
3.2 Device I/O 15
3.2.1 Unbuffered I/O to the Console 15
3.2.2 Unbuffered I/O to Non-Console Devices 16
4. Console I/O Overview 17
4.1 Line-oriented input 17
4.2 Character-oriented input 18
4.3 Using ioctl 19
4.4 The sgtty fields 19
4.5 Examples 20
5. Dynamic Buffer Allocation 22
6. Error Processing Overview 23

System Independent Functions ..... lib
Index 5
The functions 8

Style ..... style
1. Introduction 3
2. Structured Programming 7
3. Top-down Programming 8
4. Defensive Programming and Debugging 10
5. Things to watch out for 15

Compiler Error Codes ..... err
1. Summary 4
2. Explanations 7
3. Fatal Error Messages 35

OVERVIEW

Overview

The Aztec C68k Software Development Package for the Amiga is a set of programs for developing programs in the C programming language; the resulting programs run on an Amiga.

Some of the features of Aztec C68k are:

* The full C language, as defined in the book *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, is supported.
* With some versions of Aztec C68k, several programs are provided that are similar to UNIX programs: *Z*, a full-screen text editor that is similar to the UNIX *Vi* editor; *make*, which automates some of the steps in program development and maintainance; *grep*, a pattern matcher; and *diff*, a program that determines the difference in source files.
* An extensive set of user-callable functions is provided.
* Modular programming is supported, allowing the components of a program to be compiled separately, and then linked together.
* Assembly language code can either be combined in-line with C source code, or placed in separate modules which are then linked with C modules.

There are two classes of user-callable functions: system independent and system dependent. The system-independent functions are compatible with their UNIX counterparts and with the system-independent functions provided with Aztec C packages for other systems. Use of these functions allows programs to be recompiled for use on UNIX-based systems or on other systems supported by Aztec C with little or no change.

The system-dependent functions allow programs to take advantage of special features of the Amiga.

Components

Aztec C68k contains the following components:

* *cc*, *as*, *ln*, and *lb*, the compiler, assembler, linker, and object module librarian;
* Object libraries containing user-callable functions and support functions;

* Several utility programs, including, with some versions of Aztec C68k, programs similar in function to the UNIX utilities *make*, *grep* and *diff*.

Preview

This manual is divided into two sections, each of which is in turn divided into chapters. The first section presents Amiga-specific information; the second describes features that are common to all Aztec C packages. Each chapter is identified by a symbol.

The Amiga-specific chapters and their identifying codes are:

tut describes how to get started with Aztec C68k: it discusses the installation of Aztec C68k and gives an overview of the process for turning a C source program into an executable form;

cc, *as*, and *ln* present detailed information on the compiler, assembler, and linker;

utility describes the utility programs that are provided with Aztec C68k;

libov68 describes Amiga-specific overview information;

lib68 describes the special, Aztec C68k/Amiga functions provided with Aztec C68k;

libamiga describes the functions in Aztec C68k that are used to access the Amiga function.

tech discusses several miscellaneous topics;

debug describes the debugging utilities that are provided with Aztec C68k.

The System-independent chapters and their codes are:

libov presents an overview of the functions provided with Aztec C;

lib describes the system-independent functions provided with Aztec C68k;

style discusses several topics related to the development of C programs;

err lists and describes the error messages which are generated by the compiler and linker.

TUTORIAL INTRODUCTION

Chapter Contents

Tutorial Introduction ..... tut
1. Getting Started 3
   1.1 Copying the disks 3
   1.2 Disk usage during development 4
2. Using the CLI 4
   2.1 Looking around 4
   2.2 Working with files 6
3. Compiling, assembling, and linking 7
4. Environment variables 8
5. The ram disk 8
6. Where to go from here 9

Tutorial Introduction

Congratulations on purchasing the Manx Aztec C68K development system for the Amiga. You're probably anxious to get started, so this chapter will provide a short tutorial introduction to the system.

We will first describe how to make backup copies of the distribution disks. We will then talk a little bit about the CLI environment and some things we've done to enhance it. Finally, we go through the steps you can follow to create and execute the 'hello world!' program.

1. Getting Started

The first thing you should do with your Aztec C68K software is to make a copy of the distribution disks. There is no copy protection, so the copies can be used for development, while the originals should be stored in a safe place.

1.1 Copying the disks

First off, make sure that the write protect tab on the master disks are in the no-write position so that we can't hurt them even by accident. Next, if you have just booted your machine, insert the Kickstart disk and wait till it asks for the WorkBench disk.

If you are already running the WorkBench, remove your disk and while holding down the control key, press the two Amiga keys simultaneously. This will force the machine to perform a warm boot and you should see the insert WorkBench screen.

Instead of your WorkBench disk, insert the Aztec C disk labeled number 1. This disk will boot directly into the CLI and will display a startup message something like:

```
Welcome to the Wonderful World of Aztec C!!
        Version 3.2x
         02/20/86
```

Then it will ask for the current date and time:

```
Date (MM/DD/YY)?
Time (HH:MM:SS)?
```

Respond to each in the specified format. Then there will be a prompt that looks like:

```
1>
```

This is the standard CLI prompt. It is used to indicate that the CLI is

waiting for a command to be typed by the user.

To copy the disks, enter the following command:

diskcopy df0: to df1:

If you only have one disk drive, the command should be:

diskcopy df0: to df0:

After typing the appropriate command, press the RETURN key to load the DISKCOPY program from the disk. The DISKCOPY program will prompt you to place the appropriate disks in the drives and to press RETURN when ready. Using blank disks follow the directions and make a copy of each of the disks you received. If you want more information about the DISKCOPY program, it can be found in the AmigaDOS User's Manual from Commodore.

1.2 Disk usage during development

While developing programs for the Amiga, you will normally use two disks. The first disk (the one we booted), contains the compiler, assembler, linker, editor, and the most common CLI commands. It also contains the Amiga system files, the Aztec and Amiga C header files and the Aztec libraries.

Thus, this disk contains all the files necessary to develop your program. The second disk will contain the source and object modules that you will be creating and working with.

To create an empty disk suitable for use as a work disk, use the FORMAT command in the following way:

format drive df1: name "WorkDisk"

You can name the disk whatever you like, you don't have to use the name "WorkDisk". The FORMAT program will then ask you to insert your blank disk and press RETURN when ready.

If you only have a single drive, there will be some spare room for creating files. However, to get the maximum space, refer to the chapter on single-drive use in the Technical Information chapter.

2. Using the CLI

In this section, we will introduce the CLI more officially and talk briefly about some of the more important commands. You should still have the boot disk in the drive and the CLI prompt on the screen.

2.1 Looking around

The first thing we need to do is get some information, so naturally we type the following command to the CLI:

info

This command will display information about the disks which are currently in the disk drives. This will tell you exactly how much space is free on each of your disks.

The Amiga supports what is known as a "hierarchical" file structure that is used to organize files on the disk. An analogy may help to explain this.

Think of a disk as a closet where we want to store different sorts of items. Each disk we have can be thought of as a different closet. If we put all the items into a closet, it gets pretty cluttered and hard to find that baseball we wanted. So to be more organized, we can let our closet have some doors that lead to other closets. Now we can have a door labeled "sports" and another labeled "clothes". We can still save things in this closet as well.

If we go into the "sports" closet, we can have all our sports equipment there, or we can have some more doors in this closet that are labeled "baseball", "football", and "tennis". This could go on for quite a bit, but you should get the idea. The construction of the house would probably be a real challenge, but with the Amiga, we can do create exactly this kind of organization on our disks.

The name we use when talking about a "closet" on the disk is a "directory". For example, type the command:

```
dir
```

This is short for "DIRectory" and will display the contents of the current directory or closet that we are in. You will notice that the first set of files displayed are followed by the word "(dir)". This is short for "directory" and indicates that there are some additional closet doors in the current closet. The names with the "(dir)" following them are the items stored here.

Let's go into one of the other closets by typing:

```
cd include
```

This is short for "Change Directory". Now we are in the new directory and another "dir" command will show us what we have in here. We find some more "closets" and some more items.

If we're not sure where we are at any time, just typing:

```
cd
```

by itself will get our current location printed. The first thing will be master closet or directory that we are in. This will usually be "df0:" or "df1:" depending on which drive we are using. This is followed by the name of each "sub-closet door" or sub-directory that we went through to reach the current one. Each name will be separated from the preceding by a '/' character.

We can go anyplace we like by using the CD command and typing the full name of the closet or directory we want. For example, to go into the "C" directory that we saw when we did the first DIR command, just type:

```
cd df0:c
```

If the closet doesn't exist, an error message will be displayed.

Try doing a DIR in this directory and you will notice a lot of files and you will also notice that some of the files have familiar names like CD, DIR, FORMAT and DISKCOPY. This is the command directory and is where the CLI gets all the commands that it executes.

Now let's go someplace else. Type:

```
cd df0:include/exec
```

which will place us in the *include* directory on the df0: disk. Instead of DIR try typing:

```
list
```

This command is like dir, but gives more information about each file and directory in the current directory. Now let's take a quick look at one of the files here. The command

```
type errno.h
```

will display the contents of the file "errno.h" on the screen of the Amiga.

The commands INFO, CD, DIR, LIST and TYPE are the most often used commands when moving and looking around in the CLI. Try looking in other directories as well.

2.2 Working with files

But, now, let's try and do something useful. First, create a new directory by first moving to the top or "root" directory. This is done by typing:

```
cd df0:
```

Next, let's make ourselves a directory to put our files in and keep them out of the way. We do this by typing:

```
makedir test
```

which creates a new directory named "test" in the current directory. Go into "test" by typing:

```
cd test
```

Now let's create a simple program using the COPY command by typing:

```
copy * to hello
```

The '*' tells the copy command to get the source from the keyboard and the "to" says to put what it gets into the file "hello". The disk will whir and the cursor will wait for input without a prompt. Now, type the following short program:

```
main()
{
printf("hello world!!\n");
}
```

To tell the copy command we are done, hold down the CTRL key and then press the '\' key. You should see the CLI prompt once again.

To make sure the file is correct, use the TYPE command again to display the file to the screen. If you now do a DIR command, you will see the file "hello". Since most C program source files are distinguished by a file name extension of ".c", use the following command to rename the file:

```
rename hello hello.c
```

This changes the name of the first file to the second.

A much better way to create and modify files is to use a text editor. The Amiga computer comes with two editors which are described in the AmigaDOS User's Manual. In addition, the Commercial version of Aztec C comes with an editor named 'Z', which is similar to the UNIX editor 'vi'.

3. Compiling, assembling and linking

Now that we have our file, let's make an executable version that we can run. To compile and automatically assemble this program, enter:

```
cc hello.c
```

This will invoke the Aztec C68K compiler to compile the file "hello.c". It will also automatically assemble the file as well. Do a DIR command and you will see that a new file, "hello.o", has been created. This is the object file created by the assembler.

To turn this into something we can run, we need to link with the library.

```
ln hello.o -lc
```

This invokes the linker to link the "hello.o" module with the standard C library and places the output in the file "hello". Notice how much disk activity takes place during the link phase. The linker is taking all the pieces from the library that it needs and is writing them out to the executable file.

To execute the program, just type:

```
hello
```

To get rid of the ".o" file, type:

```
delete hello.o
```

which deletes the file.

4. Environment variables

When the linker links with the libraries, or when the compiler includes header files, you have to tell them where to find them. The easiest way is to use something called an environment variable.

This is a variable with a name whose value is a string and which can be located by any program that wants to. Environment variables are created with the SET command. To see what environment variables currently exist, just type:

```
set
```

with no arguments and it will display the current environment.

You will notice that two items have been already defined, CLIB and INCLUDE. These were created by a command in a file whose full name is "df0:s/Startup-Sequence". The command in the file looked like:

```
set CLIB=df0:lib/ INCLUDE=df0:include
```

The linker uses the CLIB variable to know where to look for the libraries, while the compiler and assembler use the INCLUDE variable to find header files.

5. The ram disk

When we linked the "hello" program, it took a little while because the linker had to look all over the disk for the parts of the library that it needed and put them together to make the program. The slowest operation on the disk drives is to move around a lot between reads of the disk. The best way to speed up the linking of a program is to speed up the seeking process.

This can be accomplished very nicely by using a very fast disk or by using a ram disk. The Amiga computer comes with a built-in ram disk that lies dormant until you want to use it. Since the ram disk uses up memory that is also needed for programs to run, you need at least 512K of memory to use it.

To make the link run faster, we need to first copy the library over to the ram disk. We do this by typing:

copy df0:lib/c.lib ram:

Now, we need to get the linker to look in the right place, so we type:

set CLIB=ram:

which changes the environment variable used by the linker.

Now if we type:

ln hello.o -lc

there should be a lot less disk activity, and it should be a lot faster.

6. Where to go from here

In this chapter, we've just begun to describe the features of the full Aztec C68K development environment. You should know enough now to create some simple programs, which you can do while continuing to read the rest of this manual. There are also a number of public domain programs that we have collected from various sources to serve as examples. Check the release document for a list and more information.

In your reading, be sure to check the sections on the compiler and linker. You should scan through the Utility Programs chapter which describes in detail the programs provided with the various versions of Aztec C68K. If you aren't already familiar with the AmigaDOS CLI commands and options, it would also be well worth your while to read the AmigaDOS User's Manual.

Once you are accustomed to writing C programs with Aztec C68K, you can start writing programs that access the special features of the Amiga. For this, read the Amiga functions chapter of this manual, which provides a list of all the Amiga functions and their arguments. To use these functions, it is best to purchase the ROM Kernel and Intuition manuals which describe all the functions in detail as well as providing examples of their use.

To access the Amiga functions, a program must *#include* header files that define Amiga data structures, constants, and variables. You can significantly decrease the compile time of programs that call Amiga functions by specially compiling these header files, and then including the resultant "precompiled" header file in your programs instead of the unprocessed header files. For more information, see the discussion of *#include* files in the Operator Information section of the Compiler chapter.

Finally, we hope you enjoy using Aztec C and find it a productive environment in which to program this outstanding machine!

THE COMPILER

Chapter Contents

The compiler cc
1. Operating Instructions 3
1.1 The C Source File 3
1.2 The Output Files 3
1.3 *#include* files 5
1.4 Memory Models 7
2. Compiler Options 13
2.1 Summary of Options 13
2.2 Description of Options 15
3. Programmer Information 19
3.1 Supported Language Features 19
3.2 Structure Assignment 19
3.3 Structure Passing 19
3.4 Line Continuation 19
3.5 The *void* Data Type 19
3.6 Special Symbols 20
3.7 String Merging 20
3.8 Long Names 21
3.9 Reserved Words 21
3.10 Global Variables 21
3.11 Data Formats 21
3.12 Register Usage 22
3.13 In-line Assembly Language Code 23
3.14 Writing Machine-Independent Code 23
4. Error Processing 26

The Compiler

This chapter describes *cc*, the Aztec C compiler. It is not intented to be a complete guide to the C language; for that, you must consult other texts. One such text is *The C Programming Language*, by Kernighan and Ritchie. The compilers were implemented according to the language description in the Kernighan and Ritchie book.

This description of the compilers is divided into four subsections, which describe how to use the compiler, compiler options, information related to the writing of programs, and error processing.

1. Compiler Operating Instructions

cc is invoked by a command of the form:

```
cc [-options] filename.c
```

where [-options] specify optional parameters, and *filename.c* is the name of the file containing the C source program. Options can appear either before or after the name of the C source file.

The compiler reads C source statements from the input file, translates them to assembly language source, and writes the result to another file.

1.1 The C source file

The extension on the source file name is optional. If not specified, it's assumed to be *.c*. For example, with the following command, the compiler will assume the file name is *text.c*:

```
cc text
```

1.2 The output files

1.2.1 Creating an object code file

Normally, when you compile a C program you are interested in the relocatable object code for the program, and not in its assembly language source. Because of this, the compiler by default writes the assembly language source for a C program to an intermediate file and then automatically starts the assembler. The assembler then translates the assembly language source to relocatable object code, writes this code to a file, and erases the intermediate file.

By default, the object code generated by a *cc*-started assembly is sent to a file whose name is derived from that of the file containing the C source by changing its extension to *.o*. This file is placed in the directory that contains the C source file. For example, if the compiler

is started with the command

```
cc prog.c
```

the file *prog.o* will be created, containing the relocatable object code for the program.

The name of the file containing the object code created by a compiler-started assembler can also be explicitly specified when the compiler is started, using the compiler's -O option. For example, the command

```
cc -O myobj.rel prog.c
```

compiles and assembles the C source that's in the file *prog.c*, writing the object code to the file *myobj.rel*.

When the compiler is going to automatically start the assembler, it by default writes the assembly language source to a temporary file named *ctmpxxx.xxx*, where the x's are replaced by digits in such a way that the name becomes unique. This temporary file is placed in the directory specified by the environment variable *CCTEMP*. If this variable doesn't exist, the file is placed in current directory.

When *CCTEMP* exists, the complete name of the temporary file is generated by simply prefixing its value to the ctmpxxx.xxx name. For example the first command that follows sets *CCTEMP* so that the temporary file is placed in the root directory on the *ram:* volume; the second causes it to be placed in the *compile/temp* directory on the current volume; and the third causes it to be placed in the *tmp* directory on the *df1:* drive.

```
set CCTEMP=ram:
set CCTEMP=:compile/temp/
set CCTEMP=df1:tmp/
```

Note that a terminating backslash is required when a directory is explicitly specified, but not when just the drive is specified.

For a description on the setting of environment variables, see the Tutorial chapter and the description of the *set* command in the Utility Programs chapter.

If you are interested in the assembly language source, but still want the compiler to start the assembler, specify the option -T when you start the compiler. This will cause the compiler to send the assembly language source to a file whose name is derived from that of the file containing the C source by changing its extension to *.asm*. The C source statements will be included as comments in the assembly language source. For example, the command

```
cc -T prog.c
```

compiles and assembles *prog.c*, creating the files *prog.asm* and *prog.o*.

1.2.2 Creating just an assembly language file

There are some programs for which you don't want the compiler to automatically start the assembler. For example, you may want to modify the assembly language generated by the compiler for a particular program. In such cases, you can use the compiler's *-A* option to prevent the compiler from starting the assembler.

When you compile a program using the *-A* option, you can tell the compiler the name and location of the file to which it should write the assembly language source, using the *-O* option.

If you don't use the *-O* option but do use the *-A* option, the compiler will send the assembly language source to a file whose name is derived from that of the C source file by changing the extension to *.asm* and place this file in the same directory as the one that contains the C source file. For example, the command

```
cc -A prog.c
```

compiles, without assembling, the C source that's in *prog.c*, sending the assembly language source to *prog.asm*.

As another example, the command

```
cc -A -O temp.asm prog.c
```

compiles, without assembling, the C source that's in *prog.c*, sending the assembly language source to the file *temp.asm*.

When the -A option is used, the option -T causes the compiler to include the C source statements as comments in the assembly language source.

1.3 #*include* Files

1.3.1 Searching for #*include* files

You can make the compiler search for #*include* files in a sequence of directories, thus allowing source files and #*include* files to be contained in different directories.

Directories can be specified with the -I compiler option, and with the INCLUDE environment variable. The compiler itself also selects a few areas to search. The maximum number of searched areas is eight.

If the file name in the #include statement specifies a directory, just that directory is searched.

1.3.1.1 The -I option.

A -I option defines a single directory to be searched. The area descriptor follows the -I, with no intervening blanks. For example, the following *-I* option tells the compiler to search the *include* directory on the *ram:* volume:

-Iram:include

1.3.1.2 The INCLUDE environment variable

The INCLUDE environment variable also defines directories to be searched for #include files. The string associated with this variable consists of the names of the directories to be searched, with each pair of names separated by a semicolon. For example, with *INCLUDE* set by the following command, the compiler will search for include files in the directories *cc:include*, *df0:hdr*, and *ram:include*:

set INCLUDE=cc:include;df0:hdr;ram:include;

For information on environment variables, see the Tutorial chapter and the description of the *set* command in the Utility Programs chapter.

1.3.1.3 The search order for include files

Directories are searched in the following order:

1. If the #include statement delimited the file name with the double quote character, ", the current directory on the default drive is searched. If delimited by angle brackets, < and >, this area isn't automatically searched.
2. The directories defined in -I options are searched, in the order listed on the command line.
3. The directories defined in the INCLUDE environment variable are searched, in the order listed.

1.3.2 Precompiled #*include* Files

To shorten compilation time, the compiler supports precompiled #*include* files.

To use this feature, you first compile frequently-used header files, specifying the *+h* option; this causes the compiler to write its symbol table, which contains information about the contents of the header files, to a disk file. Then, when you compile a module that #*include*s some of these header files, you specify the *+i* option; this causes the compiler to load into its symbol table the pre-compiled symbol table information about the header files. When the compiler encounters a #*include* statement of a header file for which it has already loaded pre-compiled symbol table information, it ignores the #*include* statement. This ignoring occurs even if the #*include* file was nested within another #*include* file in the C source from which the pre-compiled symbol table was generated.

The compiler does much less work when it loads pre-compiled information into its symbol table than when it generates the same information from C source, and hence using pre-compiled #*include* files can considerably shorten the time required to compile a module.

The *+H* option tells the compiler to write its symbol table to a file. The name of the file immediately follows the *+H*, with no intervening spaces. For example, you might create a file named *x.c* that consists just of *#include* statements for all the header files that you want pre-compiled. You could then generate a file named *include.pre* that contains the symbol table information for these header files by entering the following command:

```
cc +Hinclude.pre x.c
```

The *+I* option tells the compiler to read pre-compiled symbol table information from a file. The name of the file immediately follows the *+I*, with no intervening spaces. For example, to compile the file *prog.c* that accesses the header files that were defined in *x.c*, and to have the compiler preload the symbol table information for these files from *include.pre*, enter the following command:

```
cc +Iinclude.pre prog.c
```

1.4 Memory Models

The memory model used by a program determines how the program's executable code makes references to code and data. This in turn indirectly determines the amount of code and data that the program can have, the size of the executable code, and the program's execution speed.

Before getting into the details of memory models, we want to describe the sections into which a C68k-generated program is organized. The sections of a program are these:

* *code*, containing the program's executable code;
* *data*, containing its global and static data;
* *stack*, containing its automatic variables, control information, and temporary variables;
* *heap*, an area from which buffers are dynamically allocated.

There are two attributes to a program's memory model: one attribute specifies whether the program uses the *large data* or the *small data* memory model; the other attribute specifies whether the program uses the *large code* or *small code* memory model.

1.4.1 *large data* versus *small data*

The fundamental difference between a *large data* and a *small data* program concerns the way that instructions access data segment data: a *large data* program accesses the data using position-dependent instructions; a *small data* program accesses the data using position-independent instructions. An instruction makes position-dependent reference to data in the data segment by specifying the absolute address of the data; it makes a position-independent reference to data in the data segment by specifying the location as an offset from a reserved address register. Other differences in *large data* and *small*

data programs result from this fundamental difference; these other differences are:

* There is no limit to the amount of global and static data that a *large data* program can have. A *small data* program, on the other hand, can have at most 64k bytes of global and static data.
* For a *small data* program, an address register must be reserved to point into the middle of the data segment. For a *large data* program, an instruction that wants to access data in the data segment contains the absolute address of the data, and hence doesn't need this address register.
* It takes more time to load a code segment for a *large data* program than for a *small data* program. The reason for this is that the absolute address of data segment data isn't known until a program is loaded. Thus, instructions that access data segment data using absolute addresses must be adjusted when the code segment containing the instructions is loaded, whereas instructions that access data segment data in a position-independent way don't need to be adjusted.
* A code segment is larger when its program uses *large data* than when it uses *small data*, because a reference to data in a data segment occupies a 32-bit field in a *large data* instruction, and occupies a 16-bit field in a *small data* instruction.
* A program is slower when it uses *large data* than when it uses *small data*, because it takes more time for an instruction to access data when it specifies the absolute address of the data than when it specifies the data's offset offset from an address register.

1.4.2 *large code* versus *small code*

The fundamental difference between a *large code* and a *small code* program concerns the way that instructions in the program refer to locations that are located in the code segment: for a *large code* program the reference is made using position-dependent instructions; for a *small code* program, the reference is made using position-independent instructions. An instruction makes position-dependent reference to a code segment location by specifying the absolute address of the location; it makes a position-independent reference to a code segment location by specifying the location as an offset from the current program counter. Other differences in *large data* and *small data* programs result from this fundamental difference; these other differences are:

* The size of a code segment is unlimited for both *large code* and *small code* programs. An instruction in a *large code*

program can directly call or jump to the location, regardless of its location in the code segment.

An instruction in a *small code* program can only directly call or jump to locations that are within 32k bytes of the instruction. To allow instructions in *small code* programs to transfer control to any location, regardless of its location in the code segment, a "jump table", which is located in the program's data segment, is used. If a location to which an instruction wants to transfer control is more than 32k bytes from the instruction, the transfer is made indirectly, via the jump table: the instruction calls or jumps to an entry in the jump table, which in turn jumps to the desired location. A jump instruction in a jump table entry refers to a code segment location using an absolute, 32-bit address, and hence can directly access any location in the program's code segment.

When a *small code* program is linked, the linker automatically builds the jump table: if the location to which an instruction wants to transfer control is outside the instruction's range, the linker creates a jump table entry that jumps to the location and transforms the pc-relative instruction into a position-independent call or jump to the jump table entry.

* A code segment can contain data as well as executable code. An instruction in a *large code* program can access data located anywhere in the code segment, because it accesses code segment data using position-dependent instructions, in which the location is referred to using a 32-bit, absolute address. An instruction in a *small code* program can only access code segment data that is located within 32k bytes of the instruction.

* For a *small code* program to access the jump table, an address register needs to be reserved and set up to point into the middle of the program's data segment; if the program also uses *small data*, the same address register is used for both jump table accesses and normal accesses of data segment data. For a *large code* program, this address register is not needed for the referencing of locations in the code segment.

* A program takes longer to load if it uses *large code* than if it uses *small code*. Instructions in a *large code* program that reference a code segment location must be adjusted when the program is loaded, since such instructions must contain the absolute address of the location and since this isn't known until the program is loaded. Instructions in a *small code* program that reference code segment locations need not be

adjusted, since they are always independent of the location at which the code segment is loaded: if the location is within 32k of the referencing instruction, the instruction is pc-relative; and if it's outside this range, the instruction is a position-independent jump to a jump table entry.

When a *small code* program that contains a jump table is loaded, its jump table entries must be adjusted, since these are jump instructions to code segment locations, where each instruction must contain the absolute address of the destination address. However, it should take less time to adjust the jump table for a *small code* program than to adjust the code segment of a *large code* version of the same program, since for any destination of a jump or call instruction a *small code* version of a program will have at most one jump table entry needing adjustment, whereas a *large code* version of the program may have many jump or call instructions to the same location that need to be adjusted.

* A code segment is larger when its program uses *large code* than when it uses *small code*, because instructions that reference code segment locations by specifying an absolute address use a 32-bit field to define the location, whereas instructions that reference data by specifying a pc-relative address or an offset from an index register use a 16-bit field to define the location.

* A program is usually slower when it uses *large code* than when it uses *small code*, because it takes more time for an instruction to reference a code segment location when it specifies the absolute address of the data than when it specifies the location in a pc-relative form.

A large *small code* program that has lots of indirect transfers of control via the jump table may not differ much in execution time from a *large code* version of the same program, since the *small code* indirect transfer via the jump table will take more time than the *large code* direct transfer.

1.4.3 Selecting a module's memory model

You define the memory model to be used by a module when you compile the module, by specifying or not specifying the following options:

+C	Module uses *large code*. If this option isn't specified, the module will use *small code*.
+D	Module uses *large data*. If this option isn't specified, the module will use *small data*.

For example, the following commands compile *prog.c* to use different memory models:

cc prog	small code, small data
cc +C prog	large code, small data
cc +D prog	small code, large data
cc +C +D	large code, large data

1.4.4 Libraries

The modules in the Aztec C68k libraries use the *small code*, *small data* memory model.

1.4.5 Multi-module programs

The modules that you link together to form an executable program can use different memory models, with the following caveat.

When large data and small data modules are linked together, the linker will create an arbitrarily large data segment, without attempting to sort the data into those that are accessed by large data modules and those that are accessed by small data modules. When the program is running, address register A4, which the small data modules use to access data, will point into the middle of this data segment.

Here's the caveat: data that the small data modules attempt to access must be within 32k bytes of the location pointed at by A4. The linker will detect data accesses by small data modules for which this condition isn't satisfied, and issue a message. If you get this message, try reordering the order in which the linker encounters them; if that doesn't solve the problem, you'll have to recompile the small data modules, making them use large data.

As we mentioned above, the modules in the libraries supplied with Aztec C68k use the small data memory model; hence, recompilation of a program's small data modules to use large data must include the library modules.

1.4.6 Code segmentation

By default, the linker creates a program that has a single code segment. As you've seen, there is no limit on the size of this segment, regardless of the memory model used by the program. However, there is a limit on the amount of memory in a machine. Also, the larger a program's code segment, the worse its performance: if the program uses *small code*, its execution speed will degrade as the program gets larger, because more transfers of control will have to be done indirectly, via the jump table. And if it uses *large code*, it will be larger and slower than a *small code* version of the same program.

The Aztec linker allows you to divide a program's code into multiple segments. When the program is running, code segments are brought into memory as needed; when a code segment is no longer needed in memory, the memory it occupied can be released and reused, either for another code segment or for some other purpose.

So one obviouse advantage to partitioning a large program into code segments is that it allows a program to be larger than available memory. Another advantage is that it allows the code to use *small code* memory model without incurring the degradation that is caused by lots of indirect transfers of control via the jump table.

Of course, there is some degradation of performance of a segmented program compared to a non-segmented version of the same program, since the segments must be loaded into memory from disk before they can be executed. But with juditious partitioning of the program, this degradation can be minimized.

For more information on segmented programs, see the Linker and Technical Information chapters.

2. Compiler Options

There are two types of options in Aztec C compilers: machine independent and machine dependent. The machine-independent options are provided on all Aztec C compilers. They are identified by a leading minus sign.

The Aztec C compiler for each target system has its own, machine-dependent, options. Such options are identified by a leading plus sign.

The following paragraphs first summarize the compiler options and then describe them in detail.

2.1 Summary of options

2.1.1 Machine-independent Options

-A Don't start the assembler when compilation is done.

-Dsymbol[=value]
Define a symbol to the preprocessor.

-Idir Search the directory named *dir* for #include files.

-O file Send output to *file*.

-S Don't print warning messages.

-T Include C source statements in the assembly code output as comments. Each source statement appears before the assembly code it generates.

-B Don't pause after every fifth error to ask if the compiler should continue. See the Errors subsection for details.

-Enum Use an expression table having *num* entries.

-Lnum Use an local symbol table having *num* entries.

-Ynum Use an case table having *num* entries.

-Znum Use a literal table having *num* bytes.

2.1.2 Special Options for the Amiga

+B Don't generate the statement "public .begin"

+C Generate code that uses the "large code" memory model. For information on +C and the related +D option, see the Operator Information section.

+D Generate code that uses the "large data" memory model.

+Hfile Write symbol table to *file*. For information on +H and the related +I option, see the Operator Information section.

+Ifile	Read pre-compiled symbol table from *file*.
+L	*int* variables and constants are 32 bits long. If this option isn't used, they are 16 bits wide.
+Q	Put character string constants in the data segment. If +Q isn't specified, string constants are placed in the code segment.

2.2 Detailed description of the options

2.2.1 Machine-independent options

2.2.1.1 The -D Option (Define a macro)

The *-D* option defines a symbol in the same way as the preprocessor directive, *#define*. Its usage is as follows:

```
cc -Dmacro[=text] prog.c
```

For example,

```
cc -DMAXLEN=1000 prog.c
```

is equivalent to inserting the following line at the beginning of the program:

```
#define  MAXLEN  1000
```

Since the *-D* option causes a symbol to be defined for the preprocessor, this can be used in conjunction with the preprocessor directive, *#ifdef*, to selectively include code in a compilation. A common example is code such as the following:

```
#ifdef  DEBUG
    printf("value: %d\n", i);
#endif
```

This debugging code would be included in the compiled source by the following command:

```
cc -dDEBUG program.c
```

When no substitution text is specified, the symbol is defined to have the numerical value 1.

2.2.1.2 The -I Option (Include another source file)

The *-I* option causes the compiler to search in a specified directory for files included in the source code. The name of the directory immediately follows the -I, with no intervening spaces. For more details, see the Compiler Operating Instructions, above.

2.2.1.3 The -S Option (Be Silent)

The compiler considers some errors to be genuine errors and others to be possible errors. For the first type of error, the compiler always generates an error message. For the second, it generates a warning message. The *-S* option causes the compiler to not print warning messages.

2.2.1.4 The Local Symbol Table and the -L Option

When the compiler begins processing a compound statement, such as the body of a function or the body of a *for* loop, it makes entries about the statement's local symbols in the local symbol table, and

removes the entries when it finishes processing the statement. If the table overflows, the compiler will display a message and stop.

By default, the local symbol table contains 40 entries. Each entry is 26 bytes long; thus by default the table contains 1040 bytes.

You can explicitly define the number of entries in the local symbol table using the *-L* option. The number of entries immediately follows the *-L*, with no intervening spaces. For example, the following compilation will use a table of 75 entries, or almost 2000 bytes:

```
cc -L75 program.c
```

2.2.1.5 The Expression Table and the -E Option

The compiler uses the expression table to process an expression. When the compiler completes its processing of an expression, it frees all space in this table, thus making the entire table available for the processing of the next expression. If the expression table overflows, the compiler will generate error number 36, "no more expression space", and halt.

By default, the expression table contains 80 entries. Each entry is 14 bytes long; thus by default the table contains 1120 bytes.

You can explicitly define the number of entries in the expression table using the *-E* option. The number of entries immediately follows the *-E*, with no intervening spaces. For example, the following compilation will use a table of 20 entries:

```
cc -E20 program.c
```

2.2.1.6 The Case Table and the -Y Option

The compiler uses the case table to process a switch statement, making entries in the table for the statement's cases. When it completes its processing of a switch statement, it frees up the entries for that switch. If this table overflows, the compiler will display error 76 and halt.

For example, the following will use a maximum of four entries in the case table:

```
switch (a) {
case 0:                    /* one */
   a += 1;
   break;
case 1:                    /* two */
   switch (x) {
   case 'a':               /* three */
       func1 (a);
       break;
   case 'b':               /* four */
       func2 (b);
       break;
   }                  /* release the last two */
   a = 5;
case 3:                  /* total ends at three */
   func2 (a);
   break;
}
```

By default, the table contains 100 entries. Each entry is four bytes long; thus by default, the table occupies 400 bytes.

You can explicitly define the number of entries in the case table using the compiler's *-Y* option. The number of entries immediately follows the *-Y*, with no intervening spaces. For example, the following compilation uses a case table having 50 entries:

```
cc -Y50 file
```

2.2.1.7 The String Table and the -Z Option

When the compiler encounters a "literal" (that is, a character string), it places the string in the literal table. If this table overflows, the compiler will display error 2, "string space exhausted", and halt.

By default, the literal table contains 2000 bytes.

You can explicitly define the number of bytes in this table using the compiler's *-Z* option. The number of bytes immediately follows the *-Z*, with no intervening spaces. For example, the following command will reserve 3000 bytes for the string table:

```
cc -Z3000 file
```

2.2.1.8 The Macro/Global Symbol Table

The compiler stores information about a program's macros and global symbols in the Macro/Global Symbol Table. This table is located in memory above all the other tables used by the compiler. Its size is set after all the other tables have been set, and hence can't be set by you. If this table overflows, the compiler will display the message "Out of Memory!" and halt. You must recompile, using smaller sizes for the other tables.

2.2.2 Amiga Options

2.2.2.1 The +B Option

Normally when compiling modules, the compiler generates a reference to the entry point named *.begin*. Then when modules are linked into a program, the reference causes the linker to include in the program the library module that contains *.begin*.

The *+B* option prevents the compiler from generating this reference.

For example, if you want to provide your own entry point for a program, and its name isn't *.begin*, you should compile the program's modules with the *+B* option. If you don't, then the program will be bigger than necessary, since it will contain your entry point module and the standard entry point module. In addition, the linker by default sets at the program's base address a jump instruction to the program's entry point; if it finds entry points in several modules, it will set the jump to the last one encountered.

2.2.2.2 The +C and +D options

These options are discussed in the first section of this chapter.

2.2.2.3 The +L option

The +L option causes a program's *int* variables and constants to be 32 bits long, instead of the 16 bit default length. This option has no effect on the length of a module's other integer variables: variables of type *short* and *long* are always 16 and 32 bits long, respectively.

The main use of this option is to allow Lattice C-compiled programs, which use 32 bit *int*s, to be recompiled without change with Aztec C68k.

The +L option also provides a means for integer variables and constants to be passed between the Amiga functions, for which such values are 32 bits long, and use modules. For more information on this, see the Amiga Functions chapter.

We recommend that you use the +L option sparingly, if at all, because it makes a program larger and slower.

3. Writing programs

The previous sections of this description of the compiler discussed operational features of the compiler; that is, presented information that an operator would use to compile a C program. In this section, we want to present information of interest to those who are actually writing programs.

3.1 Supported Language Features

Aztec C supports the entire C language as defined in *The C Programming Language* by Kernighan and Ritchie. This now includes the bit field data type.

The following paragraphs describe features of the standard C language that are supported by Aztec C but that aren't described in the K & R text.

3.2 Structure assignment

Aztec C supports structure assignment. With this feature, a program can cause one structure to be copied into another using the assignment operator.

For example, if *s1* and *s2* are structures of the same type, you can say:

```
s1 = s2;
```

thus causing the contents of structure s1 to be copied into structure s2.

Unlike other operators, the assignment operator doesn't have a value when it's used to copy a structure. Thus, you can't say things like "a = b = c", or "(a=b).fld" when a, b, and c are structures.

3.3 Structure Passing

Aztec C68k allows structure to be passed from one function to another function; but a function cannot return a structure as its value.

3.4 Line continuation

If the compiler finds a source line whose last character is a backslash, \, it will consider the following line to be part of the current line, without the backslash. For example, the following statements define a character array containing the string "abcdef":

```
char array[]="ab\
cd\
ef";
```

3.5 The *void* data type

Functions that don't return a value can be declared to return a *void*. This provides a safety check on the use of such functions: if a *void* function attempts to return a value, or if a function tries to use the

value returned by a *void* function, the compiler will generate an error message.

Variables can be declared to point to a *void*, and functions can be declared as returning a pointer to a *void*.

Unlike other pointers, a pointer to a *void* can be assigned to a pointer to any type of object, and vice versa. For other types of pointers, the compiler will generate a warning message if an attempt is made to assign one pointer to another, when the types of objects pointed at by the two pointers differ.

That is, the compiler will generate a warning message for the assignment statement in the following program:

```
main()
{
   char *cp;
   int *ip;
   ip = cp;
}
```

The compiler won't complain about the following program:

```
main()
{
   char *cp;
   void *getbuf();
   cp = getbuf();
}
```

3.6 Special symbols

Aztec C supports the following symbols:

___FILE___	Name of the file being compiled. This is a character string.
___LINE___	Number of the line currently being compiled. This is an integer.
___FUNC___	Name of the function currently being compiled. This is a character string.

In case you can't tell, these symbols begin and end with two underscore characters.

For example,

```
printf("file= %s\n", ___FILE___);
printf("line= %d\n", ___LINE___);
printf("func=%s\n", ___FUNC___);
```

3.7 String merging

The compiler will merge adjacent character strings. For example,

```
printf("file=" __FILE__ " line= %d func= " __FUNC__,
    __LINE__);
```

3.8 Long names

Symbol names are significant to 31 characters. This includes external symbols, which are significant to 31 characters throughout assembly and linkage.

3.9 Reserved words

const, *signed*, and *volatile* are reserved keywords, and must not be used as symbol names in your programs.

3.10 Global variables

The standard C language requires that in the modules that want to access a global variable, exactly one module declare it without the *extern* keyword and all others declare it with the *extern* keyword. Aztec C supports the following modified version of the rule:

* Multiple modules can declare the same variable, with the *extern* keyword being optional;
* When several modules declare a variable without using the *extern* keyword, the amount of space reserved for the variable is set to the largest size specified by the various declarations;
* When one module declares a variable using the *extern* keyword, at least one other module must declare the variable without using the *extern* keyword;
* At most one module can specify an initial value for a global variable;
* When a module specifies an initial value for a global variable, the amount of storage reserved for the variable is set to the amount specified in the declaration that specified an initial value, regardless of the amounts specified in the other declarations.

3.11 Data formats

3.11.1 char

Variables of type *char* are one byte long, and can be signed or unsigned. By default, a *char* variable on the Amiga is signed.

When a signed char variable is used in an expression, it's converted to a 16-bit integer by propagating the most significant bit. Thus, a char variable whose value is between 128 and 255 will appear to be a negative number if used in an expression.

When an unsigned char variable is used in an expression, it's converted to a 16-bit integer in the range 0 to 255.

A character in a *char* is in ASCII format.

3.11.2 pointer

Pointer variables are four bytes long.

3.11.3 short

Variables of type *short* are two bytes long. They can be signed or unsigned, and by default are signed.

A negative value is stored in two's complement format. A *short* is stored in memory with its least significant byte at the highest numbered address. A -2 stored at location 100 would thus look like:

location	*contents in hex*
100	FF
101	FE

3.11.4 long

Variables of type *long* occupy four bytes, and can be signed or unsigned.

Negative values are stored in two's complement format. Longs are stored sequentially with the most significant byte stored at the lowest memory address and the least significant byte at the highest memory address.

3.11.5 int

int variables are normally 16 bits long, but are 32 bits long if the *+L* compiler option is used. For more information, see the discussion of the +L option in the Options section of this chapter.

3.11.6 float & double

float and *double* numbers are both represented using the single precision format defined by the Motorola fast floating point package. However, the compiler still allocates 64 bits of storage for doubles even though only half is used. The same is true when passing a floating point number to another function.

3.12 Register Usage

Registers D0-D3 and A2-A3 are available for storage of register variables.

Registers D4-D7 are used storage of intermediate values during expression evaluation.

Register A4 is reserved for use by modules that use the small code and/or small data memory model; for such modules, this register points at the program's jump table. For more information, see the Program Organization section of the Technical Information chapter.

Register A5 points at the 'frame' on the stack that contains information about the currently-executing function.

Registers A0, A1, and A6 are used for temporary storage.

Register A7 points to the top of the program's stack.

3.13 In-Line Assembly Language Code

Assembly language source can be included in a C program, by surrounding the assembly language code with the preprocessor directives *#asm* and *#endasm*.

When the compiler encounters a *#asm* statement, it copies lines from the C source file to the assembly language file that it's generating, until it finds a *#endasm* statement. The *#asm* and *#endasm* statements are not copied.

While the compiler is copying assembly language source, it doesn't try to process or interpret the lines that it reads. In particular, it won't perform macro substitution.

A program that uses *#asm ...#endasm* must avoid the following placing in-line assembly code immediately following an *if* block; that is, it should avoid the following code:

```
if (...){
   ...
}
#asm
  ...
 #endasm
  ...
```

The code generated by the compiler will test the condition and if false branch to the statement following the *#endasm* instead of to the beginning of the assembly language code. To have the compiler generate code that will branch to the beginning of the assembly language code, you must include a null statement between the end of the *if* block and the *asm* statement:

```
if (...){
   ...
}
;
#asm
  ...
 #endasm
  ...
```

3.14 Writing machine-independent code

The Aztec family of C compilers are almost entirely compatible. The degree of compatibility of the Aztec C compilers with v7 C, system 3 C, system 5 C, and XENIX C is also extremely high. There are, however, some differences. The following paragraphs discuss

things you should be aware of when writing C programs that will run in a variety of environments.

If you want to write C programs that will run on different machines, don't use bit fields or enumerated data types, and don't pass structures between functions. Some compilers support these features, and some don't.

3.14.1 Compatibility Between Aztec Products

Within releases, code can be easily moved from one implementation of Aztec C to another. Where release numbers differ (i.e. 1.06 and 2.0) code is upward compatible, but some changes may be needed to move code down to a lower numbered release. The downward compatibility problems can be eliminated by not using new features of the higher numbered releases.

3.14.2 Sign Extension For Character Variables

If the declaration of a *char* variable doesn't specify whether the variable is signed or unsigned, the code generated for some machines assumes that the variable is signed and others that it's unsigned. For example, none of the 8 bit implementations of Aztec C sign extend characters used in arithmetic computations, whereas all 16 bit implementations do sign extend characters. This incompatibility can be corrected by declaring characters used in arithmetic computations as unsigned, or by AND'ing characters used in arithmetic expressions with 255 (0xff). For instance:

```
char a=129;
int b;
b = (a & 0xff) * 21;
```

3.14.3 The MPU... symbols

To simplify the task of writing programs that must have some system dependent code, each of the Aztec C compilers defines a symbol which identifies the processor on which the compiler-generated code will run. These symbols, and their corresponding processors, are:

symbol	*processor*
MPU68000	68000
MPU8086	8086/8088
MPU80186	80186/80286
MPU6502	6502
MPU8080	8080
MPUZ80	Z80

Only one of these symbols will be defined for a particular compiler.

Other pre-defined symbols define the exact machine on which code generated by a compiler will run. These symbols, and their

corresponding machines, are:

MCH_AMIGA	Amiga
MCH_MACINTOSH	Macintosh
MCH_ATARI	Atari
MCH_ROM	Rom-based system

For example, the following program fragment contains several machine-dependent blocks of code. When the program is compiled for execution on a particular processor, just one of these blocks will be compiled: the one containing code for that processor.

```
#ifdef MPU68000
   /* 68000 code */
#else
#ifdef MPU8086
   /* 8086 code */
#else
#ifdef MPU8080
   /* 8080 code */
#endif
#endif
#endif
```

4. Error checking

Compiler errors come in two varieties-- fatal and not fatal. Fatal errors cause the compiler to make a final statement and stop. Running out of memory and finding no input are examples of fatal errors. Both kinds of errors are described in the *Errors* chapter. The non-fatal sort are introduced below.

The compiler will report any errors it finds in the source file. It will first print out a line of code, followed by a line containing the up-arrow (caret) character. The up-arrow in this line indicates where the compiler was in the source line when it detected the error. The compiler will then display a line containing the following:

* The name of the source file containing the line;
* The number of the line within the file;
* An error code;
* The symbol which caused the error, when appropriate.

The error codes are defined and described in the *Errors* chapter.

The compiler writes error messages to its standard output. Thus, error messages normally go to the console, but they can be associated with another device or file by redirecting standard output in the usual manner. For example,

cc prog	errors sent to the console
cc prog >outerr	errors sent to the file *outerr*

The compiler normally pauses after every fifth error, and sends a message to its standard output asking you want to continue. The compiler will continue only if you enter a line beginning with the character 'y'. If you don't want the compiler to pause in this manner, (if, for example, the compiler's standard output has been redirected to a file) specify the *-B* option when you start the compiler.

The compiler is not always able to give a precise description of an error. Usually, it must proceed to the next item in the file to ascertain that an error was encountered. Once an error is found, it is not obvious how to interpret the subsequent code, since the compiler cannot second-guess the programmer's intentions. This may cause it to flag perfectly good syntax as an error.

If errors arise at compile time, it is a general rule of thumb that the very first error should be corrected first. This may clear up some of the errors which follow.

The best way to attack an error is first to look up the meaning of the error code in the back of this manual. Some hints are given there as to what the problem might be. And you will find it easier to understand the error and the message if you know why the compiler produced that particular code. The error codes indicate what the compiler was doing when the error was found.

THE ASSEMBLER

Chapter Contents

The Assembler ..... as
1. Operating Instructions 3
1.1 The Input File 3
1.2 The Object Code File 4
1.3 Listing File 4
1.4 Optimizations 4
1.5 Searching for *include* Files 4
2. Assembler Options 6
3. Programmer information 8

The Assembler

The *as* assembler translates assembly language source statements into relocatable object code. Assembler source statements are read from an input text file and the object code is written to an output file. A listing file is written if requested. The relocatable object code must be linked by *ln*, the Manx Linker, before it can be executed. At linkage time it may be combined with other object files and run time library routines from system or private libraries. Object modules produced from C source text and Assembler source text can be combined at linkage time into a composite module.

Assembly language routines are generally not required when programming in C. Assembly language routines should only be necessary where critical execution time or critical size requirements exist. Some system interfacing or low level routines may also require assembler code.

Information on the MC68000 architecture and instructions can be found in the *Motorola MC68000 16-bit Microprocessor User's Manual* (Prentice-Hall, Inc., Englewood Cliffs, N. J. 07632)

1. Operating Instructions

The assembler is started by entering the command line:

```
as [-options] filename
```

where *[-options]* specify optional parameters and *filename* is the name of the file to be assembled.

The assembler reads assembly source statements from the input file, writes the translated relocatable object code to an output file, and if requested writes a listing to an output file. The assembler also will merge assembly code from other files on encountering an *include* directive.

1.1 The Input File

Specification of the extension on the source file name is optional: if not given, it's assumed to be *.asm*. For example the following command assembles the file *io.asm*. The input file is a text file that will usually be created

```
as io
```

1.2 The Object Code File

The object code produced by the assembler is written to a file. By default, this file is placed in the directory that contains the source file, and its name is derived from that of the input file by changing the extension to *.o*.

To write the object code to another file, use the *-o* option. For example, the following command assembles the source that's in *prog.asm*, sending the object code to the file *new.obj*. This latter file is placed in the current directory, since the *-o* option didn't specify otherwise.

```
as -o new.obj prog.asm
```

1.3 Listing File

If the *-L* option is specified, the assembler will produce a listing file with the same root as the input file and a filename extension of *.lst*. The listing file displays the source statements and their machine language equivalent. The listing also indicates the relative displacement of each machine instruction.

1.4 Optimizations

The assembler by default performs some optimizations on an assembly language source file, making just two passes through the assembly source file. Optimization can be disabled using the *-N* option; this causes the asssembler to run faster, since it makes just a single pass through the source and since it needn't optimize the code, but it makes the resultant code larger and slower.

The instructions affected by these optimizations are:

branches	Long branches are converted to short if possible, and branches to the following location will be deleted.
movem	If there are no registers, the instruction is deleted. If there is only one register, the shorter *move* instruction is substituted.
jsr	*bsr* is substituted if possible.

To make these optimizations, the assembler uses a dynamically-allocated table. If this table is filled, the assembler will continue, will generate correct, but not completely optimized, object code, and will tell you the number of additional entries that it could have used. You can then recompile the module using the *-S* option to define a different table size.

1.5 Searching for *include* Files

By default the assembler searches just the current directory for files specified in *include* statements. Using the *-I* option and the INCLUDE environment variable, you can make the assembler also search other

directories for such files, thus allowing program source files and header files to be contained in different directories.

If the file name on the *include* directive specifies a directory or a drive name, the assembler will automatically search just the specified directory for the file.

1.5.1 The -I option

The *-I* option defines a single directory to be searched for a file specified in a *include* statement. The path descriptor follows the *-I*, with no intervening blanks. For example, the specification

```
as -isys:db/include prog1
```

directs the assembler to search the *sys:db/include* directory when looking for an *include* file.

Multiple *-I* options can be specified when the assembler is started, if desired, thus defining multiple directories to be searched.

1.5.2 The INCLUDE Environment Variable

The INCLUDE environment variable also defines areas to be searched for include files. The value of the variable consists of the names of the directories to be searched, with each pair of names separated by semicolons. The value of the INCLUDE environment variable is defined using the *set* utility program. For example, the following command defines three areas to be searched:

```
set INCLUDE=df0:include;ram:source/include;df1:
```

1.5.3 Include Search Order

When the assembler encounters a *include* statement, it searches directories for the file specified in the statement in the following order:

1. The current directory is searched.
2. The directories specified in the *-I* options are searched, in the order listed on the line that started the assembler;
3. The directories specified in the INCLUDE environment variable are searched.

2. Assembler Options

2.1 Summary of options

-O *filename*	Send object code to *filename*.
-I*area*	Defines an area to be searched for files specified in an *include* statement.
-L	Generate listing.
-N	Don't optimize object code.
-S*num*	Create squeeze table having *num* entries.
-V	Verbose option. Generate memory usage statistics.
-ZAP	This option is used primarily when you assemble a file that was generated by the compiler. It directs the assembler to delete the input file after processing.
-C	Make *large code* the default code memory model. If this option isn't specified, *small code* is the default code memory model. The *near code* and *far code* directives can be used by a program to override the default code memory model.
-D	Make *large data* the default data memory model. If this option isn't specified, *small data* is the default data memory model. The *near data* and *far data* directives can be used by a program to override the default data memory model. For more information on memory models, see the Compiler chapter. For more information on the *near* and *far* directives, see the Programmer Information of this chapter.
-E*name[=val]*	Create an entry in the symbol table for *name* and assign it the constant value *val*. If *val* isn't specified, *name* is assigned the value 1.

2.2 Description of options

2.2.1 The '-O *filename*' option

This option causes *as* to send the object code to *filename*. If this option isn't specified, *as* sends the object code to a file whose name is derived from that of the assembler source file by changing the extension to *.o*; in this case, the file is placed in the directory containing the source file.

2.2.2 The -I Option

The *-I* option causes the assembler to search in a specified area for files included in the source code.

The name of the area immediately follows the -I, with no intervening spaces. For example, the following defines directory /source/inc on volume *sys:* search area:

```
-Isys:/source/inc
```

For more details, see the Assembler Operating Instructions, above.

2.2.3 The -L option

Causes *as* to generate a listing. The name of the file to which the listing is sent is derived from that of the source file by changing the extension to *.lst*. The listing file is placed in the directory containing the source file.

2.2.4 The -S option

The *-S* option defines the number of entries in the squeeze table. If this option isn't specified, the table contains 1000 entries.

The number of entries immediately follows the *-S*, with no intervening spaces. For example, the following option tells the assembler to use a squeeze table containing 1050 entries:

```
-s1050
```

3. Programmer Information

The following sections discuss the four types of assembly language statements:

1. Comments
2. Instructions
3. Directives
4. Macro Calls

3.1 Comments

A comment can appear after a semicolon or after the operand field. For example:

```
; this is a comment

    link a6,#.2    this is also a comment
```

3.2 Executable Instructions

Executable instructions have the general format:

```
label operation operand
```

Labels

Assembler labels can be any length. External labels are only significant for the first 32 characters. Any additional characters will be ignored. Valid label characters include letters, numbers, or the special characters . and _. A label cannot begin with a digit.

Labels that do not start in the first column require a colon suffixed.

Operations

The assembler recognizes all of the mnemonics found in Motorola's *16-bit Microprocessor User's Manual.*

To specify a length for instructions which support multiple lengths, it is sufficient to suffix the instruction mnemonic with:

.B Specifies a length of one byte
.W Specifies a length of 16-bits
.L specifies a length of 32-bits

Operands

The operand field consists of one expression, or two expressions separated by a comma with no imbedded spaces. An expression is comprised of register mnemonics, symbols, constants, or arithmetic combinations of symbols or constants.

Symbols or labels represent relocatable or absolute values. An absolute value is one whose value is known at assembly time. A relocatable value is one whose value is not known until the program is actually loaded into memory for execution.

Relocatable expressions can only be expressed arithmetically as sums or differences. The difference between two relocatable expressions is absolute. The result of summing two relocatable expressions is undefined.

There are five type of constants: octal, binary, decimal, hexadecimal and string. An octal constant is expressed as an @ followed by a string of digits from the set 0 through 7 such as *@123* or *@777*. A binary constant is expressed as a % followed by a string of ones and zeroes such as *%10101* or *%11001100*. A decimal constant is a string of numbers. A hexadecimal constant is a *$* followed by a string of characters made up of numbers or alphabetics from *a* through *f* such as *$ffff* or *1a2e*. A string constant is any string of characters enclosed in single quotes such as *'abdc'*.

Register mnemonics include the data register mnemonics *D0* through *D7*, the address registers *A0 through A7*, *SP* or *A7* the stack pointer, *PC* the program counter (forces PC relative mode), *SR* the status register, the condition code register *CCR*. And the user stack pointer *USP*.

The assembler supports addition (+), subtraction (-), multiplication(*), division (/), shift right (>>), shift left (<<), unary minus, and (&), or (|). The order of precedence is innermost parenthesis, unary minus, shift, and/or, multiplication/division, and addition/subtraction.

3.3 Directives

The following paragraphs describe the directives that are supported by the assembler.

EQU

label equ <expression>

This directive assigns the value of the expression on the right to the label on the left.

REG

label reg <register list>

This directive assigns the value of the register list to the label. Forward references are not allowed. A register list consists of a list of register names separated by the / character. The - character may be used to identify an inclusive set of registers. The following are valid register lists:

a0-a3/d0-d2/d4
a1/a2/a4/a6/d0-d2

PUBLIC

[label] public <symbol>[,<symbol>...]

This directive identifies the specified symbols as having external scope. These symbols are visable to the linker and are used to resolve references between modules. The type of the symbol is CODE if it was defined within the code segment, DATA if it was defined within the data segment, and ABS if it was defined to have an absolute value in an *equ* directive.

GLOBAL and BSS

[label] global <symbol>,<size>
[label] bss <symbol>,<size>

These directives restore storage for uninitialized data items. The area is reserved in the uninitialized data area. If *global* is used then the data item is known to other modules that are external to the routine. If *bss* is used then the data item is local to the routine in which it is defined.

If a *global* is defined in more than one module then the linkage editor will reserve the maximum value of those assigned.

A symbol that appears in both a *global* and a *public* directive is located in the initialized data area and the global statements size parameters are ignored.

ENTRY

[label] entry <symbol>

This directive defines the entry point of the program. Only one entry can be declared per program. If no entry point is defined, the first instruction of the first module becomes the default entry point.

END

This directive defines the end of the source statements. All files are closed and the assembler terminates.

CSEG

Assembled output following this directive is output into the code segment of the program output file.

DSEG

Assembled output following this directive is placed in the

initialized data segment of the program file.

DC - Define Constant

[label]	*dc.b*	*<value>[,<value>, <value> ...]*
[label]	*dc*	*<value>[,<value>, <value> ...]*
[label]	*dc.w*	*<value>[,<value>, <value> ...]*
[label]	*dc.l*	*<value>[,<value>, <value> ...]*
[label]	*dc.b*	*"string"*

The *dc* directive causes one or more fields of memory to be allocated and initialized.

Each <value> operand causes one field to be allocated and then to be initialized with the specified value. A <value> can be an expression. An expression may contain forward references.

For command programs, a value can contain a reference to a memory location whose address won't be known until the program is loaded into memory. In this case, an item for this value will be added to the program's relocation table; when the program is loaded, the field containing this value will be set to the correct value.

Each field for a particular *dc* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

The last form listed above for *dc* allocates a field having exactly the number of characters in the string, and places the string in it.

DCB - Define Constant Block

[label]	*dcb.b*	*<size>[,<value>]*
[label]	*dcb*	*<size>[,<value>]*
[label]	*dcb.w*	*<size>[,<value>]*
[label]	*dcb.l*	*<size>[,<value>]*

The *dcb* directive allocates a block of storage containing <size> fields, and initializes each field with <value>. If <value> isn't specified, it's assumed to be 0.

Each field for a particular *dcb* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

DS - Define Storage

[label]	*ds.b*	*<size>*
[label]	*ds*	*<size>*
[label]	*ds.w*	*<size>*
[label]	*ds.l*	*<size>*

This directive allocates a block of storage containing <size> fields, and sets each field to 0.

Each field for a particular *ds* directive is the same length. A period followed by b, w, or l can be appended to a directive, defining the field length to be one, two, or four bytes, respectively. If the field length isn't specified in this way, it defaults to 2 bytes.

Fields that are two or four bytes long are aligned on word boundaries.

NEAR and FAR

near	*code/data*
far	*code/data*

The *near code* and *far code* directives cause the assembler to generate code that uses the *small code* or *large code* memory model, respectively. If these options aren't specified, the assembler will generate code whose code memory model is determined by the presence or absence of the +C assembler option.

The *near data* and *far data* directives cause the assembler to generate code that uses the *small data* or *large data* memory model, respectively. If these options aren't specified, the assembler will generate code whose data memory model is determined by the presence or absence of the +D assembler option.

A program can contain multiple *near* and *far* directives, thus allowing different sections ot the same module to use different memory models.

LIST and NOLIST

The directives *list* and *nolist* turn on and off, respectively, the listing of assembly language statements to the listing file.

MLIST and NOMLIST

The directives *mlist* and *nomlist* specify whether or not the assembly language statements generated by a macro expansion should be written to the listing file.

CLIST and NOCLIST

The directives *clist* and *noclist* specify whether or not statements should be included in the listing file, when the statements were not assembled as a result of assembler conditional statements. By default, such statements are not listed.

INCLUDE

include <file>

This directive causes the assembler to suspend assembly of the current file and to assemble the specified file. When done, the assembler continues assembling the original file.

MACRO and ENDM

[label] macro <symbol>
...
text
...
endm

The specified symbol is entered in the assembler opcodes table. The text between the *macro* and *endm* is saved in memory. When the macro symbol is encountered as an opcode the text is placed in line. Up to nine arguments can be specified. They are referenced in the macro text as %1 through %9. In expanding a macro symbolic argument references are replaced by their actual value.

MEXIT

Upon encountering this directive expansion of the current macro stops and the assembler scans for the statement following the ENDM directive.

IF, ELSE, and ENDC

if <test>
...
[else]
...
endc

These directives are used to allow conditional assembly of parts of the input file. The general form of the IF test is:

<exp>		
<exp> == <exp>	||	*<exp> = <exp>*
<exp> != <exp>	||	*<exp> <> <exp>*
'str1' == 'str2'	||	*'str1' = 'str2'*
'str1' != 'str2'	||	*'str1' <> 'str2'*

If the test result is true, then the lines up to an *ELSE* or *ENDC* are assembled. If there is an *ELSE*, then lines up to the *ENDC* are skipped. The skipped lines are not displayed in the listing file unless the *CLIST* directive has been used. If the test is false, then lines are skipped until an *ELSE* or *ENDC* is encountered. If it is an *ELSE*, then the following lines up to an *ENDC* are assembled.

An undefined symbol is treated as having the value 0.

THE LINKER

Chapter Contents

The Linker ..... ln
1. Introduction to linking 3
2. Using the Linker 7
3. Linker Options 9

The Linker

The *ln* linker has two functions:

* It ties together the pieces of a program which have been compiled and assembled separately;
* It converts the linked pieces to a format which can be loaded and executed.

The pieces must have been created by the Manx assembler.

The first section of this chapter presents a brief introduction to linking and what the linker does. If you have had previous experience with linkage editors, you may wish to continue reading with the second section, entitled "Using the Linker." There you will find a concise description of the command format for the linker.

1. Introduction to linking

Relocatable Object Files

The object code produced by the assembler is "relocatable" because it can be loaded anywhere in memory. One task of the linker is to assign specific addresses to the parts of the program. This tells the operating system where to load the program when it is run.

Linking hello.o

It is very unusual for a C program to consist of a single, self-contained module. Let's consider a simple program which prints "hello, world" using the function, *printf*. The terminology here is precise; *printf* is a function and not an intrinsic feature of the language. It is a function which you might have written, but it already happens to be provided in the file, *c.lib*. This file is a library of all the standard i/o functions. It also contains many support routines which are called in the code generated by the compiler. These routines aid in integer arithmetic, operating system support, etc.

When the linker sees that a call to *printf* was made, it pulls the function from the library and combines it with the "hello, world" program. The link command would look like this:

```
ln hello.o c.lib
```

When *hello.c* was compiled, calls were made to some invisible support functions in the library. So linking without the standard library will cause some unfamiliar symbols to be undefined.

All programs will need to be linked with one of the versions of *c.lib*. Initially, you can use *c.lib* itself. Later on, if you find that *c.lib* doesn't suit your requirements, you can use one of the other versions. For more details, see the Libraries section of the Technical Information chapter.

The Linking Process

Since the standard library contains only a limited number of general purpose functions, all but the most trivial programs are certain to call user-defined functions. It is up to the linker to connect a function call with the definition of the function somewhere in the code.

In the example given below, the linker will find two function calls in file 1. The reference to *func1* is "resolved" when the definition of *func1* is found in the same file. The following command

```
ln file1.o c.lib
```

will cause an error indicating that *func2* is an undefined symbol. The reason is that the definition of *func2* is in another file, namely *file2.o*. The linkage has to include this file in order to be successful:

```
ln file1.o file2.o c.lib
```

file 1

```
main()
{
  func1();
  func2();
}

func1()
{
  return;
}
```

file 2

```
func2()
{
return;
}
```

Libraries

A library is a collection of object files put together by a librarian. Libraries intended for use with *ln* must be built with the Manx librarian, *lb*. This utility is described in the Utility Programs chapter.

All the object files specified to the linker will be "pulled into" the linkage; they are automatically included in the final executable file. However, when a library is encountered, it is searched. Only those modules in the library which satisfy a previous function call are pulled in.

For Example

Consider the "hello, world" example. Having looked at the module, *hello.o*, the linker has built a list of undefined symbols. This list

includes all the global symbols that have been referenced but not defined. Global variables and all function names are considered to be global symbols.

The list of undefined's for *hello.o* includes the symbol *printf*. When the linker reaches the standard library, this is one of the symbols it will be looking for. It will discover that *printf* is defined in a library module whose name also happens to be *printf*. (There is not any necessary relation between the name of a library module and the functions defined within it.)

The linker pulls in the *printf* module in order to resolve the reference to the *printf* function.

Files are examined in the order in which they are specified on the command line. So the following linkages are equivalent:

```
ln hello.o

ln c.lib hello.o
```

Since no symbols are undefined when the linker searches *c.lib* in the second line, no modules are pulled in. It is good practice to leave all libraries at the end of the command line, with the standard library last of all.

The Order of Library Modules

For the same reason, the order of the modules within a library is significant. The linker searches a library once, from beginning to end. If a module is pulled in at any point, and that module introduces a new undefined symbol, then that symbol is added to the running list of undefined's. The linker will not search the library twice to resolve any references which remain unresolved. A common error lies in the following situation:

module of program	*references (function calls)*
main.o	getinput, do_calc
input.o	gets
calc.o	put_value
output.o	printf

Suppose we build a library to hold the last three modules of this program. Then our link step will look like this:

```
ln main.o proglib.lib c.lib
```

But it is important that *proglib.lib* is built in the right order. Let's assume that *main()* calls two functions, *getinput()* and *do_calc()*. *getinput()* is defined in the module *input.o*. It in turn calls the standard library function *gets()*. *do_calc()* is in *calc.o* and calls *put_value()*. *put_value()* is in *output.o* and calls *printf()*.

What happens at link time if *proglib.lib* is built as follows?

```
proglib.lib:        input.o
                    output.o
                    calc.o
```

After *main.o*, the linker has *getinput* and *do_calc* undefined (as well as some other support functions in *c.lib*). Then it begins the search of *proglib.lib*. It looks at the library module, *input*, first. Since that module defines *getinput*, that symbol is taken off the list of undefined's. But *gets* is added to it.

The symbols *do_calc* and *gets* are undefined when the linker examines the module, *output*. Since neither of these symbols are defined there, that module is ignored. In the next module, *calc*, the reference to *do_calc* is resolved but *put_value* is a new undefined symbol.

The linker still has *gets* and *put_value* undefined. It then moves on to *c.lib*, where *gets* is resolved. But the call to *put_value* is never satisfied. The error from the linker will look like this:

```
Undefined symbol: _put_value
```

This means that the module defining *put_value* was not pulled into the linkage. The reason, as we saw, was that *put_value* was not an undefined symbol when the *output* module was passed over. This problem would not occur with the library built this way:

```
proglib.lib:        input.o
                    calc.o
                    output.o
```

The standard libraries were put together with much care so that this kind of problem would not arise.

Occasionally it becomes difficult or impossible to build a library so that all references are resolved. In the example, the problem could be solved with the following command:

```
ln main.o proglib.lib proglib.lib c.lib
```

The second time through *proglib.lib*, the linker will pull in the module *output*. The reason this is not the most satisfactory solution is that the linker has to search the library twice; this will lengthen the time needed to link.

2. Using the Linker

The general form of a linkage is as follows:

```
ln [-options] file1.o [file2.o ...] [lib1.lib ...]
```

The linker combines object modules produced by the *as* assembler into an executable program. It can search libraries of object modules for functions needed to complete the linkage; including just the needed modules in the executable program. The linker makes just a single pass through a library, so that only forward references within a library will be resolved.

The executable file

The name of the executable output file can be selected using the -O linker option. If this option isn't used, the linker will derive the name of the output file from that of the first object file listed on the command line, by deleting its extension. In the default case, the executable file will be located in the directory in which the first object file is located. For example,

```
ln prog.o c.lib
```

will produce the file *prog*. The standard library, *c.lib*, will have to be included in most linkages.

A different output file can be specified with the -O option, as in the following command:

```
ln -o program mod1.o mod2.o c.lib
```

This command also shows how several individual modules can be linked together. A "module", in this sense, is a section of a program containing a limited number of functions, usually related. These modules are compiled and assembled separately and linked together to produce an executable file.

Libraries

Several libraries of object modules are provided with Aztec C68k. The most frequently-used of these are *c.lib*, which contains the non-floating point functions, and *m.lib*, which contains the floating point functions. Other libraries are provided with some versions of Aztec C68k; for their description, see the Libraries section of the Technical Information chapter.

All programs must be linked with one of the versions of *c.lib*. In addition to containing all the non-floating point functions described in the Functions chapter, it contains internal functions which are called by compiler-generated code.

Programs that perform floating point operations must be linked with one of the versions of *m.lib*, in addition to a version of *c.lib*. The floating point library must be specified on the linker command line

before *c.lib*.

Libraries of user modules can also be searched by the linker. These are created with the Manx *lb* program, and must be listed on the linker command line before the Manx libraries.

For example, the following links the module *program.o*, searching the libraries *mylib.lib*, *new.lib*, *m.lib*, and *c.lib* for needed modules:

```
ln program.o mylib.lib new.lib m.lib c.lib
```

Each of the libraries will be searched once in the order in which they appear on the command line.

Libraries can be conveniently specified using the *-L* option. For example, the following command is equivalent to the following:

```
ln -o program.o -lmylib -lnew -lm -lc
```

For more information, see the description of the *-L* option in the Options section of this chapter.

3. Linker Options

3.1 Summary of options

-O file	Write executable code to the file named *file.*
-Lname	Search the library *name.lib* for needed modules.
-F file	Read command arguments from *file.*
-T	Generate an ASCII symbol table file.
-W	Generate a Wack-readable symbol table file.
-V	Be verbose.
+O[i]	Place the executable code for the object modules that follow in code segment *i.* If *i* isn't specified, use the first empty segment. If the segment already exists, append the code to its end. For more information, see the *Code Segmentation* section of the Technical Information chapter.
+C[cdb]	Force loading into chip memory (ie, standard, built-in memory) of the program sections whose identifying letters follow the +C.
+F[cdb]	Force loading into fast memory (ie, external, add-on memory) of the program sections whose identifying letters follow the +F.

3.2 Detailed description of the options

The -O option

The *-O* option can be used to specify the name of the file to which the linker is to write the executable program. The name of this file is in the parameter that follows the *-O*. For example, the following command writes the executable program to the file *progout*:

```
ln -o progout prog.o c.lib
```

If this option isn't used, the linker derives the name of the executable file from that of the first input file, by deleting its extension.

The -L option

The *-L* option provides a convenient means of specifying to the linker a library that it should search, when the extension of the library is *.lib*.

The name of the library is derived by prepending the string "df0:" and appending the string ".lib" to the string that follows the *-L* option. For example, with the libraries *subs.lib*, *io.lib*, *m.lib*, and *c.lib* in a directory specified by *CLIB*, you can link the module *prog.o*, and have the linker search the libraries for needed modules by entering

```
ln prog.o -lsubs -lio -lm -lc
```

The -F option

-F file causes the linker to merge the contents of the given file with the command line arguments. For example, the following command causes the linker to create an executable program in the file *myprog*. The linker includes the modules *myprog.o*, *mod1.o*, and *mod2.o* in the program, and searches the libraries *mylib.lib* and *c.lib* for needed modules.

```
ln myprog.o -f argfil c.lib
```

where the file *argfil*, contains the following:

```
mod1.o mod2.o
mylib.lib
```

The linker arguments in *argfile* can be separated by tabs, spaces, or newlines.

There are several uses for the *-F* option. The most obvious is to supply the names of modules that are frequently linked together. Since all the modules named are automatically pulled into the linkage, the linker does not spend any time in searching, as with a library. Furthermore, any linker option except *-F* can be given in a *-F* file. *-F* can appear on the command line more than once, and in any order. The arguments are processed in the order in which they are read, as

always.

The -T option

The *-T* option creates a file that contains a symbol table for the linkage, in human-readable form. This file is just a text file which lists each symbol with a hexadecimal address. This address is either the entry point for a function or the location in memory of a data item. A perusal of this file will indicate which functions were actually included in the program.

The symbol table file will have the same name as that of the file containing the executable program, with extension changed to *.sym*.

There are several special symbols which will appear in the table. They are defined in the Program Organization section of the Technical Information chapter.

The -W option

The *-W* option creates a disk file that contains a symbol table for the linkage, in the format expected by Wack.

The -V option

The *-V* option causes the linker to send a progress report of the linkage to the screen as each input file is processed. This is useful in tracking down undefined symbols and other errors which may occur while linking.

The +C and +F options

There are two types of memory on an Amiga: 'chip' memory, which is inside the Amiga; and 'fast' memory, which is external to the Amiga. By default, the three sections of a program (code, initialized data, and uninitialized data) are loaded into whatever memory is available. The +C and +F options allow you to force selected sections of a program to be loaded into chip and fast memory, respectively. Program sections are identified to an option by appending a letter to the option. These letters, and the sections to which they refer are:

c	Executable code.
d	Initialized data.
b	Uninitialized data (bss).

If an option is used without appending any letters to it, then all the program's sections are forced to be loaded into the specified type of memory.

For example:

```
ln +cdb +fc ....
```

will force a program's initialized and uninitialized data to be loaded into chip memory and the program's executable code to be loaded into

fast memory. Note that if there is no extra memory, the program will not load.

More normally, you would use:

```
ln +cdb ....
```

which will place the program's initialized and uninitialized data into chip memory and its executable code into whatever memory is available.

Finally:

```
ln +c ...
```

will force the program's initialized data, uninitialized data, and executable code to be loaded into chip memory; it's equivalent to "+ccdb".

UTILITY PROGRAMS

Chapter Contents

Utility Programs ..... util
acvt 4
adump 6
arcv 7
cmp 8
cnm 9
diff 13
grep 17
hd 23
lb 24
make 35
mkarcv 7
obd 52
ord 53
set 54
setdate 55
Z 56

Utility Programs

This chapter describes utility programs that are provided with this package.

NAME

acvt - Amiga object module converter

SYNOPSIS

acvt [-adivxz] [-t name] [-s name] file

DESCRIPTION

acvt converts the specified *file*, which contains an object module or library that has been generated by the Amiga assembler or librarian, into assembly language acceptable to the Manx assembler.

Normally, when a library is being converted, *acvt* converts all the library's modules. The *-t* and *-s* options, which are described below, can be used to convert selected modules.

acvt writes a converted module to the file whose name is derived by appending *.a68* to the module name. If this file already exists, *acvt* will say so, and allow you to enter another name; if you don't enter another name, *acvt* will overwrite the existing file.

acvt supports the following options:

option	*meaning*
a	Set asm mode. Causes the generated output to include the raw hex bytes, which then looks like an assembler listing. The generated output can NOT be used as input to the assembler. Used if there is a question about how the disassembler did something.
d	Set decimal mode. Causes generated numbers to be in decimal. If this option isn't used, generated numbers are in hex.
i	Info. Causes all symbols defined in a library to be listed along with the module name. The listing is sent to *acvt*'s stdout.
v	Verbose. Causes modules encountered in a library to be listed.
x	Extract a module. Causes *acvt* to extract specified modules from a library, in Amiga object module format. When this option is used, *acvt* does not generate assembler source.
z	Set zero mode. By default, a two byte zero at the end of a code hunk is ignored unless this flag is set.
s name	Perform the selected operations on modules, beginning with the module named *name*.

t name Perform the selected operations only on the module named *name*.

EXAMPLES

For example, to get a list of all modules and functions in a library:

```
acvt >a.lst -i amiga.lib
```

To disassemble the module containing just BeginIO, type:

```
acvt -t __BeginIO amiga.lib
```

To extract multiple modules starting with SendIO, type:

```
acvt -x -s __SendIO amiga.lib
```

NAME

adump - Amiga object module dumper

SYNOPSIS

adump [-cds] file

DESCRIPTION

adump displays information the specified *file*, which contains an object or load module that has been generated by the Amiga assembler or linker.

adump always lists the length of each of the module's blocks. The options to *adump* cause it to display the following additional information:

option	*meaning*
c	Display code block contents, in hex.
d	Display data block contents, in hex.
s	Display symbol block contents (ie, symbol names). Symbol blocks are found in executable files if debugging information has been included.

NAME

arcv & mkarcv - source dearchiver & archiver

SYNOPSIS

arcv arcfile [destpfix]

mkarcv arcfile

DESCRIPTION

arcv extracts the source from the archive *arcfile*, which has been previously created by *mkarcv*.

destpfix defines the directory in which the generated files are placed: if it's not specified, the generated files are placed in the current directory. If it is specified, it's prepended to the name of the file that *arcv* would otherwise use.

mkarcv creates the archive file *arcfile*, placing in it the files whose names it reads from its standard input. Only one file name is read from a standard input line.

EXAMPLES

For example, the file *header.arc* contains the source for all the header files. To create these header files in the current directory, enter:

```
arcv header.arc
```

The following command creates the archive *myarc.arc* containing the files *in.c*, *out.c*, and *hello.c*:

```
mkarcv myarc.arc <myarc.bld
```

The names of the following three files are contained in the file *myarc.bld*:

```
in.c
out.c
hello.c
```

NAME

cmp - File comparison utility

SYNOPSIS

cmp [-l] file1 file2

DESCRIPTION

cmp compares two files on a character-by-character basis. When it finds a difference, it displays a message, giving the offset from the beginning of the file.

If the *-l* option isn't specified, the program will stop after the first difference, displaying a message in the format:

Files differ: character 10

If the *-l* option is specified, *cmp* will list all differences, in the format:

decimal-offset hex-offset file1-valuefile2-value

EXAMPLES

cmp otst ntst

Files differ: character 10

and

cmp -l otst ntst

```
 10   a:  00 45
100   64:  1a 23
```

NAME

cnm - display object file info

SYNOPSIS

cnm [-sol] file [file ...]

DESCRIPTION

cnm displays the size and symbols of its object file arguments. The files can be object modules created by the Manx assembler, libraries of object modules created by the *lb* librarian, and, when applicable, 'rsm' files created by the Manx linker during the linking of an overlay root.

For example, the following displays the size and symbols for the object module *sub1.o* and the library *c.lib*:

```
cnm sub1.o c.lib
```

By default, the information is sent to the console. It can be redirected to a file or device in the normal way. For example, the following commands send information about *sub1.o* to the display and to the file *dispfile*:

```
cnm sub1.o
cnm sub1.o > dispfile
```

The first line listed by *cnm* for an object module has the following format:

```
file (module): code: cc   data: dd   udata: uu   total: tt (0xhh)
```

where

* *file* is the name of the file containing the module,
* *module* is the name of the module; if the module is unnamed, this field and its surrounding parentheses aren't printed;
* *cc* is the number of bytes in the module's code segment, in decimal;
* *dd* is the number of bytes in the module's initialized data segment, in decimal;
* *uu* is the number of bytes in the module's uninitialized data segment, in decimal;
* *tt* is the total number of bytes in the module's three segments, in decimal;
* *hh* is the total number of bytes in the module's three segments, in hexadecimal.

If *cnm* displays information about more than one module, it displays four totals just before it finishes, listing the sum of the sizes of the modules' code segments, initialized data segments, and uninitialized data segments, and the sum of the sizes of all segments of all modules. Each sum is in decimal; the total of all segments is also given in hexadecimal.

The *-s* option tells *cnm* to display just the sizes of the object modules. If this option isn't specified, *cnm* also displays information about each named symbol in the object modules.

When *cnm* displays information about the modules' named symbols, the *-l* option tells *cnm* to display each symbol's information on a separate line and to display all of the characters in a symbol's name; if this option isn't used, *cnm* displays the information about several symbols on a line and only displays the first eight characters of a symbol's name.

The *-o* option tells *cnm* to prefix each line generated for an object module with the name of the file containing the module and the module name in parentheses (if the module is named). If this option isn't specified, this information is listed just once for each module: prefixed to the first line generated for the module.

The *-o* option is useful when using *cnm* in combination with *grep*. For example, the following commands will display all information about the module *perror* in the library *c.lib*:

```
cnm -o c.lib >tmp
grep perror tmp
```

cnm displays information about an module's 'named' symbols; that is, about the symbols that begin with something other than a period followed by a digit. For example, the symbol *quad* is named, so information about it would be displayed; the symbol *.0123* is unnamed, so information about it would not be displayed.

For each named symbol in a module, *cnm* displays its name, a two-character code specifying its type, and an associated value. The value displayed depends on the type of the symbol.

If the first character of a symbol's type code is lower case, the symbol can only be accessed by the module; that is, it's local to the module. If this character is upper case, the symbol is global to the module: either the module has defined the symbol and is allowing other modules to access it or the module needs to access the symbol, which must be defined as a global or public symbol in another module. The type codes are:

ab The symbol was defined using the assembler's EQUATE directive. The value listed is the equated value of its symbol.

The compiler doesn't generate symbols of this type.

pg The symbol is in the code segment. The value is the offset of the symbol within the code segment.

The compiler generates this type symbol for function names. Static functions are local to the function, and so have type *pg*; all other functions are global, that is, callable from other programs, and hence have type *Pg*.

dt The symbol is in the initialized data segment. The value is the offset of the symbol from the start of the data segment.

The compiler generates symbols of this type for initialized variables which are declared outside any function. Static variables are local to the program and so have type *dt*; all other variables are global, that is, accessable from other programs, and hence have type *Dt*.

ov When an overlay is being linked and that overlay itself calls another overlay, this type of symbol can appear in the rsm file for the overlay that is being linked. It indicates that the symbol is defined in the program that is going to call the overlay that is being linked.

The value is the offset of the symbol from the beginning of the physical segment that contains it.

un The symbol is used but not defined within the program. The value has no meaning.

In assembly language terms, a type of *Un* (the U is capitalized) indicates that the symbol is the operand of a *public* directive and that it is perhaps referenced in the operand field of some statements, but that the program didn't create the symbol in a statement's label field.

The compiler generates *Un* symbols for functions that are called but not defined within the program, for variables that are declared to be *extern* and that are actually used within the program, and for uninitialized, global dimensionless arrays. Variables which are declared to be *extern* but which are not used within the program aren't mentioned in the assembly language source file generated by the compiler and hence don't appear in the object file.

bs The symbol is in the uninitalized data segment. The value is the space reserved for the symbol.

The compiler generates *bs* symbols for static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *bs* symbols for symbols defined using the *bss* assembler directive.

Gl The symbol is in the uninitialized data segment. The value is the space reserved for the symbol.

The compiler generates *Gl* symbols for non-static, uninitialized variables which are declared outside all functions and which aren't dimensionless arrays.

The assembler generates *Gl* symbols for variables declared using the *global* directive which have a non-zero size.

NAME

diff - Source file comparison utility

SYNOPSIS

diff [-b] file1 file2

DESCRIPTION

diff is a program, similar to the UNIX program of the same name, that determines the differences between two files containing text. *file1* and *file2* are the names of the files to be compared.

1. The -b option

The -b option causes *diff* to ignore trailing blanks (spaces and tabs) and to consider strings of blanks to be identical. If this option isn't specified, *diff* considers two lines to be the same only if they match *exactly.*

For example, if file1 contains the the line

```
^abc$
```

(^ and $ stand for "the beginning of the line" and "the end of the line", respectively, and aren't actually in the file) and if file2 contains the line

```
^abc  $
```

then *diff* would consider the two lines to be the same or different, depending on whether or not it was started with the -b option.

And *diff* would consider the lines

```
^a      b c$
```

and

```
^a b c$
```

to be the same or different, depending on whether or not it was started with the -b option.

diff will never consider blanks to match a null string, regardless of whether -b was used or not. So *diff* will never consider the lines

```
^abc$
```

and

```
^a bc$
```

to be the same.

2. The conversion list

diff writes, to its standard output, a "conversion list" that describes the changes that need to be made to *file1* to convert it into *file2*. The list is organized into a sequence of items, each of which describes one operation that must be performed on *file1*.

2.1 Conversion items

There are three types of operations that can be specified in a conversion list item:

* adding lines to *file1* from *file2*;
* deleting lines from *file1*;
* replacing (changing) *file1* lines with *file2* lines.

A conversion list item consists of a command line, followed by the lines in the two files that are affected by the item's operation.

2.1.1 The command line

An item's command line contains a letter describing the operation to be performed: 'a' for adding lines, 'd' for deleting lines, and 'c' for changing lines.

Preceding and following the letter are the numbers of the lines in *file1* and *file2*, respectively, that are affected by the command. If a range of lines in a file are affected, just the beginning and ending line numbers are listed, separated by a comma.

For example, the following command line says to add line 3 of *file2* after line 5 of *file1*:

```
5a3
```

and the next command line says to add lines 8,9, and 10 of *file2* after line 16 of *file1*:

```
16a8,10
```

The next command line says to delete lines 100 through 150 from *file1*, and that the last line in *file2* that matched a *file1* line was number 75:

```
100,150d75
```

The following command says to replace (change) line 32 in *file1* with line 33 in *file2*:

```
32c33
```

and the next command says to replace lines 453 through 500 in *file1* with lines 490 through 499 in *file2*:

```
453,500c490,499
```

2.1.2 The affected lines

As mentioned above, the lines affected by a conversion item's operation are listed after the item's command line. The affected lines from *file1* are listed first, flagged with a preceding '<'. Then come the affected lines from *file2*, flagged with a preceding '>'. The *file1* and *file2* lines are separated by the line

```
---
```

For example, the following conversion item says to add line 6 of *file2* after line 4 of *file1*. Line 6 of *file2* is "for (i=1; i<10;++i)":

```
4a6
> for (i=1; i<10;++i)
```

Since no lines from *file1* are affected by an 'add' conversion item, only the *file2* lines that will be added to *file1* are listed, and the separator line "---" is omitted.

The following conversion item says to delete lines 100 and 101 from *file1*, and that the last *file2* line that matched a *file1* line was numbered 110. The deleted lines were "int a;" and "double b;". Only the deleted lines are listed, and the separator line "---" is omitted:

```
100,101d110
< int a;
< double b;
```

The following conversion item says to replace lines 53 through 56 in *file1* with lines 60 and 61 in *file2*. Lines 53 through 56 in *file1* are "if (a=b){", " d = a;", " a++;", and "}". Lines 60 and 61 of *file2* are "if (a==b)" and "d = a++;".

```
53,55c60,61
< if (a=b){
<   d = a;
<   a++;
< }
---
> if (a==b)
>   d = a++;
```

3. Differences between the UNIX and Manx versions of *diff*

The Manx and UNIX versions of *diff* are actually most similar when the latter program is invoked with the -h option. As with the UNIX *diff* when used with the -h option, the Manx *diff* works best when changed stretches are short and well separated, and works with files of unlimited length.

Unlike the UNIX *diff*, the Manx *diff* doesn't support the options e, f, or h.

Unlike the UNIX *diff*, the Manx version requires that both operands to *diff* be actual files. Because of this, the Manx version of *diff* doesn't support the features of the UNIX version which allows one operand to be a directory name, (to specify a file in that directory having the same name as the other operand), and which allows one operand to be '-' (to specify *diff*'s standard input instead of a file).

NAME

grep - pattern-matching program

SYNOPSIS

grep [-cflnv] pattern [files]

DESCRIPTION

grep is a program, similar to the UNIX program of the same name, that searches files for lines containing a pattern. By default, such lines are written to *grep*'s standard output.

1. Input files

The **files** parameter is a list of files to be searched. If no files are specified, *grep* searches its standard input. Each file name can specify a single file to be searched. A name can also specify a class of files to be searched, using the special characters '*' and '?'. The character '*' matches any string of characters in a file name, and '?' matches any single character. For example,

```
grep int main.c sub1.c sub2.c
```

searches *main.c, sub1.c,* and *sub2.c* for the string *int.* The command

```
grep int *.c
```

searches all files whose extension is *.c* for the string *int.* The command

```
grep int a*.txt b*.doc
```

searches for the string *int* in each file whose (1) extension is *.txt* and *first* character is *a* and whose (2) extension is *.doc* and first character is *b.* The command

```
grep int sub?.c
```

searches for the string *int* in each file whose filename contains four characters, the first three being *sub,* and whose extension is *.c.*

2. Options

The following options are supported:

v	Print all lines that don't match the pattern.
c	Print just the name of each file and the number of matching lines that it contained.
l	Print the names of just the files that contain matching lines.
n	Precede each matching line that's printed by its relative line number within the file that contains it.
f	A character in the pattern will match both its upper and lower case equivalent.

3. Patterns

A pattern consists of a limited form of regular expression. It describes a set of character strings, any of whose members are said to be matched by the regular expression.

Some patterns match just a single character; others, which match strings, can be constructed from those that match single characters. In the following paragraphs, we'll first describe the patterns that match a single character, and then describe patterns that match strings of characters.

3.1 Matching single characters

The patterns that match a single character are these:

* An ordinary character (that is, one other than the special characters described below) matches itself.

* A period (.) is a pattern that matches any character except newline.

* A non-empty string of characters enclosed in square brackets, [], matches any one character in that string. For example, the pattern

  [ad9@]

  matches any one of the characters *a, d, 9,* or @.

  If, however, the string begins with the caret character (^), the regular expression matches any character except the other enclosed characters and newline. The '^' has this special meaning only if it is the first character of the string. For example, the pattern

  [^ad9@]

  matches any single character *except a, d, 9,* or @.

  The minus character ,-, can be used to indicate a range of consecutive ASCII characters. For example, [0-9] is equivalent to [0123456789].

* A backslash (\) followed by a special character matches the special character itself. The special characters are:

  ., *, [, and \, which are always special, except when they appear in square brackets, [].

  ^ (caret), which is special when it is at the beginning of an entire regular expression (as discussed in 3.4) and when it immediately follows the left of a pair of square brackets.

  $, which is special at the end of an entire regular

expression (discussed in 3.4).

3.2 Matching character strings

Patterns can be concatenated. In this case, the resulting pattern matches strings whose substrings match each of the concatenated patterns. For example, the pattern

```
abc
```

matches the string *abc*. This pattern is built from the patterns *a*, *b*, and *c*. The pattern

```
a.c
```

matches strings containing three characters, whose first and last characters are *a* and *c*, respectively, such as

```
abc
a@c
axc
```

3.3 Matching repeating characters

A pattern can be built by appending an asterick (*) to a pattern that matches a single character. The resulting pattern matches zero or more occurrences of the single-character pattern For example, the pattern

```
a*
```

matches any line containing zero or more *a* characters. And the pattern

```
sub[1-4]*end
```

matches lines containing strings such as

```
subend
sub132132end
```

3.4 Matching strings that begin or end lines

An entire pattern may be constrained to match only character strings that occur at the beginning or the end of a line, by beginning or ending the pattern with the character '^' or '$', respectively. For example, the pattern

```
^main
```

matches the line that begins

```
main
```

but not one that begins

```
the main ...
```

The pattern

line$

matches the line ending in

... the end of the line

but not the line ending in

a hard-hit line drive.

4. Examples

4.1 Simple string matching

The following command will search the files *file1.txt* and *file2.txt* and print the lines containing the word *heretofore*:

grep heretofore file1.txt file2.txt

If you aren't interested in the specific lines of these files, but just want to know the names of the files containing the word *heretofore*, you could enter

grep -l heretofore file1.txt file2.txt

The above two examples ignore lines in which *heretofore* contains capital letters, such as when it begins a sentence. The following command will cover this situation:

grep -lf heretofore file1.txt file2.txt

grep processes all options at once, so multiple options must be specified in one dash parameter. For example, the command

grep -l -f heretofore file1.txt file2.txt

won't work.

4.2 The special character '.'

Suppose you want to find all lines in the file prog.c that contain a four-character string whose first and last characters are 'm' and 'n', respectively, and whose other characters you don't care about. The command

grep m..n prog.c

will do the trick, since the special character '.' matches any single character.

4.3 The backslash character

There are occasions when you want to find the character '.' in a file, and don't want grep to consider it to be special. In this case, you can use the backslash character, '\', to turn off the special meaning of the next character.

For example, suppose you want to find all lines containing

.PP

Entering

grep .PP prog.doc

isn't adequate, because it will find lines such as

THE APPLICATION OF

since the '.' matches the letter 'A'. But if you enter

grep \.PP prog.doc

grep will print just what you want.

The backslash character can be used to turn off the special meaning of any special character. For example,

grep \\n prog.c

finds all lines in prog.c containing the string '\n'.

4.4 The dollar sign and the caret ($ and ^)

Suppose you want to find the number of the line on which the definition of the function *add* occurs in the file *arith.c.* Entering

grep -n add arith.c

isn't good, because it will print lines in which *add* is called in addition to the line you're interested in. Assuming that you begin all function definitions at the beginning of a line, you could enter

grep ^add arith.c

to accomplish your purpose.

The character '$' is a companion to '^', and stands for 'the end of the line'. So if you want to find all lines in *file.doc* that end in the string *time,* you could enter

grep time$ file.doc

And the following will find all lines that contain just *.PP:*

grep ^\.PP$

4.5 Using brackets

Suppose that you want to find all lines in the file *file.doc* that begin with a digit. The command

grep ^[0123456789] file.doc

will do just that. This command can be abbreviated as

grep ^[0-9] file.doc

And if you wanted to print all lines that don't begin with a digit, you could enter

```
grep ^[^0-9] file.doc
```

4.6 Repeated characters

Suppose you want to find all lines in the file *prog.c* that contain strings whose first character is 'e' and whose last character is 'z'. The command

```
grep e.*z prog.c
```

will do that. The 'e' matches an 'e', the '.*' matches zero or more arbitrary characters, and the 'z' matches a 'z'.

5. Differences between the Manx and UNIX versions of grep

The Manx and UNIX versions of *grep* differ in the options they accept and the patterns they match.

5.1 Option differences

* The option -f is supported only by the Manx *grep.*
* The options -b and -s are supported only by the UNIX *grep.*

5.2 Pattern differences

Basically, the patterns accepted by the Manx *grep* are a subset of those accepted by the UNIX *grep.*

* The Manx *grep* doesn't allow a regular expression to be surrounded by '\(' and '\)'.
* The Manx *grep* doesn't accept the construct '\{m\}'.
* The Manx *grep* doesn't allow a right bracket, ']', to be specified within brackets.
* Quoted strings can't be passed to the Manx *grep.* For example, the Manx *grep* won't accept

```
grep 'this is a fine kettle of fish' file.doc
```

NAME

hd - hex dump utility

SYNOPSIS

hd [+n[.]] file1 [+n[.]] file 2 ...

DESCRIPTION

hd displays the contents of one or more files in hex and ascii to its standard output.

file1, *file2*, ... are the names of the files to be displayed.

+n specifies the offset into the file where the display is to start, and defaults to the beginning of the file. If *+n* is followed by a period, *n* is assumed to be a decimal number; otherwise, it's assumed to be hexadecimal. Each file will be displayed beginning at the last specified offset.

EXAMPLES

hd +16b oldtest newtest +0 junk

Displays the data forks of the files *oldtest* and *newtest*, beginning at offset 0x16b, and of the file named *junk* beginning at its first byte.

hd +1000. tstfil

Displays the contents of *tstfil*, beginning at byte 1000.

NAME

lb - object file librarian

SYNOPSIS

lb library [options] [mod1 mod2 ...]

DESCRIPTION

lb is a program that creates and manipulates libraries of object modules. The modules must be created by the Manx assembler.

This description of *lb* is divided into three sections: the first describes briefly *lb*'s arguments and options, the second *lb*'s basic features, and the third the rest of *lb*'s features.

1. The arguments to *lb*

1.1 The *library* argument

When started, *lb* acts upon a single library file. The first argument to *lb* (*library*, in the synopsis) is the name of this file. The filename extension for *library* is optional; if not specified, it's assumed to be *.lib*.

1.2 The *options* argument

There are two types of *options* argument: function code options, and qualifier options. These options will be summarized in the following paragraphs, and then described in detail below.

1.2.1 Function code options

When *lb* is started, it performs one function on the specified library, as defined by the *options* argument. The functions that *lb* can perform, and their corresponding option codes, are:

function	*code*
create a library	(no code)
add modules to a library	-a, -i, -b
list library modules	-t
move modules within a library	-m
replace modules	-r
delete modules	-d
extract modules	-x
ensure module uniqueness	-u
define module extension	-e
help	-h

In the synopsis, the *options* argument is surrounded by square brackets. This indicates that the argument is optional; if a code isn't specified, *lb* assumes that a library is to be created.

1.2.2 Qualifier options

In addition to a function code, the *options* argument can optionally specify a qualifier, that modifies *lb*'s behavior as it is performing the requested function. The qualifiers and their codes are:

verbose	-v
silent	-s

The qualifier can be included in the same argument as the function code, or as a separate argument. For example, to cause *lb* to append modules to a library, and be silent when doing it, any of the following option arguments could be specified:

```
-as
-sa
-a -s
-s -a
```

1.3 The *mod* arguments

The arguments *mod1*, *mod2*, etc. are the names of the object modules, or the files containing these modules, that *lb* is to use. For some functions, *lb* requires an object module name, and for others it requires the name of a file containing an object module. In the latter case, the file's extension is optional; if not specified, the *lb* that's supplied with native Aztec C systems assumes that it's *.o*, and the *lb* that's supplied with cross development versions of Aztec C assumes that the extension is *.r*. You can explicitly define the default module extension using the *-e* option.

1.4 Reading arguments from another file

lb has a special argument, *-f filename*, that causes it to read command line arguments from the specified file. When done, it continues reading arguments from the command line. Arguments can be read from more than one file, but the file specified in a *-f filename* argument can't itself contain a *-f filename* argument.

2. Basic features of *lb*

In this section we want to describe the basic features of *lb*. With this knowledge in hand, you can start using *lb*, and then read about the rest of the features of *lb* at your leisure.

The basic things you need to know about *lb*, and which thus are described in this section, are:

* How to create a library
* How to list the names of modules in a library
* How modules get their names

* Order of modules in a library
* Getting *lb* arguments from a file

Thus, with the information presented in this section you can create libraries and get a list of the modules in libraries. The third section of this description shows you how to modify selected modules within a library.

2.1 Creating a Library

A library is created by starting *lb* with a command line that specifies the name of the library file to be created and the names of the files whose object modules are to be copied into the library. It doesn't contain a function code, and it's this absence of a function code that tells *lb* that it is to create a library.

For example, the following command creates the library *exmpl.lib*, copying into it the object modules that are in the files *obj1.o* and *obj2.o*:

```
lb exmpl.lib obj1.o obj2.o
```

Making use of *lb*'s assumptions about file names for which no extension is specified, the following command is equivalent to the above command:

```
lb exmpl obj1 obj2
```

An object module file from which modules are read into a new library can itself be a library created by *lb*. In this case, all the modules in the input library are copied into the new library.

2.1.1 The temporary library

When *lb* creates a library or modifies an existing library, it first creates a new library with a temporary name. If the function was successfully performed, *lb* erases the file having the same name as the specified library, and then renames the new library, giving it the name of the specified library. Thus, *lb* makes sure it can create a library before erasing an existing one.

Note that there must be room on the disk for both the old library and the new.

2.2 Getting the table of contents for a library

To list the names of the modules in a library, use *lb*'s *-t* option. For example, the following command lists the modules that are in *exmpl.lib*:

```
lb exmpl -t
```

The list will include some ***DIR*** entries. These identify blocks within the library that contain control information. They are created and deleted automatically as needed, and cannot be changed by you.

2.3 How modules get their names

When a module is copied into a library from a file containing a single object module (that is, from an object module generated by the Manx assembler), the name of the module within the library is derived from the name of the input file by deleting the input file's volume, path, and extension components.

For example, in the example given above, the names of the object modules in *exmpl.lib* are *obj1* and *obj2*.

An input file can itself be a library. In this case, a module's name in the new library is the same as its name in the input library.

2.4 Order in a library

The order of modules in a library is important, since the linker makes only a single pass through a library when it is searching for modules. For a discussion of this, see the tutorial section of the Linker chapter.

When *lb* creates a library, it places modules in the library in the order in which it reads them. Thus, in the example given above, the modules will be in the library in the following order:

```
obj1 obj2
```

As another example, suppose that the library *oldlib.lib* contains the following modules, in the order specified:

```
sub1    sub2    sub3
```

If the library *newlib.lib* is created with the command

```
lb newlib mod1 oldlib.lib mod2 mod3
```

the contents of the newly-created *newlib.lib* will be:

```
mod1   sub1    sub2    sub3    mod2   mod3
```

The *ord* utility program can be used to create a library whose modules are optimally sorted. For information, see its description later in this chapter.

2.5 Getting *lb* arguments from a file

For libraries containing many modules, it is frequently inconvenient, if not impossible, to enter all the arguments to *lb* on a single command line. In this case, *lb*'s *-f filename* feature can be of use: when *lb* finds this option, it opens the specified file and starts reading command arguments from it. After finishing the file, it continues to scan the command line.

For example, suppose the file *build* contains the line

```
exmpl obj1 obj2
```

Then entering the command

```
lb -f build
```

causes *lb* to get its arguments from the file *build*, which causes *lb* to create the library *exmpl.lib* containing *obj1* and *obj2*.

Arguments in a -f file can be separated by any sequence of whitespace characters ('whitespace' being blanks, tabs, and newlines). Thus, arguments in a -f file can be on separate lines, if desired.

The *lb* command line can contain multiple *-f* arguments, allowing *lb* arguments to be read from several files. For example, if some of the object modules that are to be placed in *exmpl.lib* are defined in *arith.inc*, *input.inc*, and *output.inc*, then the following command could be used to create *exmpl.lib*:

```
lb exmpl -f arith.inc -f input.inc -f output.inc
```

A -f file can contain any valid *lb* argument, except for another -f. That is, -f files can't be nested.

3. Advanced *lb* features

In this section we describe the rest of the functions that *lb* can perform. These primarily involve manipulating selected modules within a library.

3.1 Adding modules to a library

lb allows you to add modules to an existing library. The modules can be added before or after a specified module in the library or can be added to the beginning or end of the library.

The options that select *lb*'s add function are:

option	*function*
-b target	add modules before the module *target*
-i target	same as *-b target*
-a target	add modules after the module *target*
-b+	add modules to the beginning of the library
-i+	same as *-b+*
-a+	add modules to the end of the library

In an *lb* command that selects the *add* function, the names of the files containing modules to be added follows the add option code (and the target module name, when appropriate). A file can contain a single module or a library of modules.

Modules are added in the order that they are specified. If a library is to be added, its modules are added in the order they occur in the input library.

3.1.1 Adding modules before an existing module

As an example of the addition of modules before a selected module, suppose that the library *exmpl.lib* contains the modules

```
obj1    obj2    obj3
```

The command

```
lb exmpl -i obj2 mod1 mod2
```

adds the modules in the files *mod1.o* and *mod2.o* to *exmpl.lib*, placing them before the module *obj2*. The resultant *exmpl.lib* looking like this:

```
obj1    mod1    mod2    obj2    obj3
```

Note that in the *lb* command we didn't need to specify the extension of either the file containing the library to which modules were to be added or the extension of the files containing the modules to be added. *lb* assumed that the extension of the file containing the target library was *.lib*, and that the extension of the other files was *.o*.

As an example of the addition of one library to another, suppose that the library *mylib.lib* contains the modules

```
mod1    mod2    mod3
```

and that the library *exmpl.lib* contains

```
obj1    obj2    obj3
```

Then the command

```
lb -b obj2 mylib.lib
```

adds the modules in *mylib.lib* to *exmpl.lib*, resulting in *exmpl.lib* containing

```
obj1    mod1    mod2    mod3    obj2    obj3
```

Note that in this example, we had to specify the extension of the input file *mylib.lib*. If we hadn't included it, *lb* would have assumed that the file was named *mylib.o*.

3.1.2 Adding modules after an existing module

As an example of adding modules after a specified module, the command

```
lb exmpl -a obj1 mod1 mod2
```

will insert *mod1* and *mod2* after *obj1* in the library *exmpl.lib*. If *exmpl.lib* originally contained

```
obj1    obj2    obj3
```

then after the addition, it contains

```
obj1    mod1    mod2    obj2    obj3
```

3.1.3 Adding modules at the beginning or end of a library

The options *-b+* and *-a+* tell *lb* to add the modules whose names follow the option to the beginning or end of a library, respectively. Unlike the *-i* and *-a* options, these options aren't followed by the name of an existing module in the library.

For example, given the library *exmpl.lib* containing

```
obj1    obj2
```

the following command will add the modules *mod1* and *mod2* to the beginning of *exmpl.lib*:

```
lb exmpl -i+ mod1 mod2
```

resulting in *exmpl.lib* containing

```
mod1 mod2 obj1 obj2
```

The following command will add the same modules to the end of the library:

```
lb exmpl -a+ mod1 mod2
```

resulting in *exmpl.lib* containing

```
obj1    obj2    mod1    mod2
```

3.2 Moving modules within a library

Modules which already exist in a library can be easily moved about, using the *move* option, *-m*.

As with the options for adding modules to an existing library, there are several forms of *move* functions:

option	*meaning*
-mb target	move modules before the module *target*
-ma target	move modules after the module *target*
-mb+	move modules to the beginning of the library
-ma+	move modules to the end of the library

In the *lb* command, the names of the modules to be moved follows the 'move' option code.

The modules are moved in the order in which they are found in the original library, not in the order in which they are listed in the *lb* command.

3.2.1 Moving modules before an existing module

As an example of the movement of modules to a position before an existing module in a library, suppose that the library *exmpl.lib* contains

obj1 obj2 obj3 obj4 obj5 obj6

The following command moves *obj3* before *obj2*:

lb exmpl -mb obj2 obj3

putting the modules in the order:

obj1 obj3 obj2 obj4 obj5 obj6

And, given the library in the original order again, the following command moves *obj6*, *obj2*, and *obj1* before *obj3*:

lb exmpl -mb obj3 obj6 obj2 obj1

putting the library in the order:

obj1 obj2 obj6 obj3 obj4 obj5

As an example of the movement of modules to a position after an existing module, suppose that the library *exmpl.lib* is back in its original order. Then the command

lb exmpl -ma obj4 obj3 obj2

moves *obj3* and *obj2* after *obj4*, resulting in the library

obj1 obj4 obj2 obj3 obj5 obj6

3.2.2 Moving modules to the beginning or end of a library

The options for moving modules to the beginning or end of a library are *-mb+* and *-ma+*, respectively.

For example, given the library *exmpl.lib* with contents

obj1 obj2 obj3 obj4 obj5 obj6

the following command will move *obj3* and *obj5* to the beginning of the library:

lb exmpl -mb+ obj5 obj3

resulting in *exmpl.lib* having the order

obj3 obj5 obj1 obj2 obj4 obj6

And the following command will move *obj2* to the end of the library:

lb exmpl -ma+ obj2

3.3 Deleting Modules

Modules can be deleted from a library using *lb*'s *-d* option. The command for deletion has the form

lb libname -d mod1 mod2 ...

where *mod1*, *mod2*, ... are the names of the modules to be deleted.

For example, suppose that *exmpl.lib* contains

```
obj1    obj2    obj3    obj4    obj5    obj6
```

The following command deletes *obj3* and *obj5* from this library:

```
lb exmpl -d obj3 obj5
```

3.4 Replacing Modules

The *lb* option 'replace' is used to replace one module in a library with one or more other modules.

The 'replace' option has the form *-r target*, where *target* is the name of the module being replaced. In a command that uses the 'replace' option, the names of the files whose modules are to replace the target module follow the 'replace' option and its associated target module. Such a file can contain a single module or a library of modules.

Thus, an *lb* command to replace a module has the form:

```
lb library -r target mod1 mod2 ...
```

For example, suppose that the library *exmpl.lib* looks like this:

```
obj1 obj2 obj3 obj4
```

Then to replace obj3 with the modules in the files *mod1.o* and *mod2.o*, the following command could be used:

```
lb exmpl -r obj3 mod1 mod2
```

resulting in *exmpl.lib* containing

```
obj1    obj2    mod1    mod2    obj4
```

3.5 Uniqueness

lb allows libraries to be created containing duplicate modules, where one module is a duplicate of another if it has the same name.

The option *-u* causes *lb* to delete duplicate modules in a library, resulting in a library in which each module name is unique. In particular, the *-u* option causes *lb* to scan through a library, looking at module names. Any modules found that are duplicates of previous modules are deleted.

For example, suppose that the library *exmpl.lib* contains the following:

```
obj1    obj2    obj3    obj1    obj3
```

The command

```
lb exmpl -u
```

will delete the second copies of the modules *obj1* and *obj2*, leaving the library looking like this:

```
obj1    obj2    obj3
```

3.6 Extracting modules from a Library

The *lb* option *-x* extracts modules from a library and puts them in separate files, without modifying the library.

The names of the modules to be extracted follows the *-x* option. If no modules are specified, all modules in the library are extracted.

When a module is extracted, it's written to a new file; the file has same name as the module and extension *.o*.

For example, given the library *exmpl.lib* containing the modules

```
obj1 obj2 obj3
```

The command

```
lb exmpl -x
```

extracts all modules from the library, writing *obj1* to *obj1.o*, *obj2* to *obj2.o*, and *obj3* to *obj3.o*.

And the command

```
lb exmpl -x obj2
```

extracts just *obj2* from the library.

3.7 The 'verbose' option

The 'verbose' option, *-v*, causes *lb* to be verbose; that is, to tell you what it's doing.

This option can be specified as part of another option, or all by itself. For example, the following command creates a library in a chatty manner:

```
lb exmpl -v mod1 mod2 mod3
```

And the following equivalent commands cause *lb* to remove some modules and to be verbose:

```
lb exmpl -dv mod1 mod2
lb exmpl -d -v mod1 mod2
```

3.8 The 'silence' option

The 'silence' option, *-s*, tells *lb* not to display its signon message.

This option is especially useful when redirecting the output of a list command to a disk file, as described below.

3.9 Rebuilding a library

The following commands provide a convenient way to rebuild a library:

```
lb exmpl -st > tfil
lb exmpl -f tfil
```

The first command writes the names of the modules in *exmpl.lib* to the file *tfil*. The second command then rebuilds the library, using as arguments the listing generated by the first command.

The *-s* option to the first command prevents *lb* from sending information to *tfil* that would foul up the second command. The names sent to *tfil* include entries for the directory blocks, ***DIR***, but these are ignored by *lb*.

3.10 Defining the default module extension.

Specification of the extension of an object module file is optional; the *lb* that comes with native development versions of Aztec C assumes that the extension is *.o*, and the *lb* that comes with cross development versions of Aztec C assumes that it's *.r*. You can explicitly define the default extension using the *-e* option. This option has the form

```
-e .ext
```

For example, the following command creates a library; the extension of the input object module files is *.i*.

```
lb my.lib -e .i mod1 mod2 mod3
```

3.11 Help

The *-h* option is provided for brief lapses of memory, and will generate a summary of *lb* functions and options.

NAME

make - Program maintenance utility

SYNOPSIS

make [-n] [-f makefile] [-a] [name1 name2 ...]

DESCRIPTION

make is a program, similar to the UNIX program of the same name, whose primary function is to create, and keep up-to-date, files that are created from other files, such as programs, libraries, and archives.

When told to make a file, *make* first ensures that the files from which the target file is created are up-to-date or current, recreating just the ones that aren't. Then, if the target file is not current, *make* creates it.

Inter-file dependencies and the commands which must be executed to create files are specified in a file called the 'makefile', which you must write.

make has a rule-processing capability, which allows it to infer, without being explicitly told, the files on which a file depends and the commands which must be executed to create a file. Some rules are built into *make*; you can define others within the makefile.

A rule tells *make* something like this:

> "a target file having extension '.x' depends on the file having the same basic name and extension '.y'. To create such a target file, apply the commands ...".

Rules simplify the task of writing a makefile: a file's dependency information and command sequences need be explicitly specified in a makefile only if this information can't be inferred by the application of a rule.

make has a macro capability. A character string can be associated with a macro name; when the macro name is invoked in the makefile, it's replaced by its string.

Preview

The rest of this description of *make* is divided into the following sections:

1. The basics
2. Advanced features
3. Examples

1. The basics

In this section we want to present the basic features of *make*, with which you'll be able to start using *make*. Section 2 describes the other

features of *make*.

Before you can begin using *make*, you must know what *make* does, how to create a simple makefile that contains dependency entries, how to take advantage of *make*'s rule-processing capability, and, finally, how to tell *make* to make a file. Each of these topics is discussed in the following paragraphs.

1.1 What make does

The main function of *make* is to make a target file "current", where a file is considered "current" if the files on which it depends are current and if it was modified more recently than its prerequisite files. To make a file current, *make* makes the prerequisite files current; then, if the target file is not current, *make* executes the commands associated with the file, which usually recreates the file.

As you can see, *make* is inherently recursive: making a file current involves making each of its prerequisite files current; making these files current involves making each of their prerequisite files current; and so on.

make is very efficient: it only creates or recreates files that aren't current. If a file on which a target file depends is current, *make* leaves it alone. If the target file itself is current, *make* will announce the fact and halt without modifying the target.

> **It is important to have the time and date set for *make* to behave properly, since *make* uses the 'last modified' times that are recorded in files' directory entries to decide if a target file is not current.**

1.2 The makefile

When *make* starts, one of the first things it does is to read a file, which you must write, called the 'makefile'. This file contains dependency entries defining inter-file dependencies and the commands that must be executed to make a file current. It also contains rule definitions and macro definitions.

In the following paragraphs, we want to just describe dependency entries. In section 2 we discuss the somewhat more advanced topics of rule and macro definition.

A dependency entry in a makefile defines one or more target files, the files on which the targets depend, and the operating system commands that are to be executed when any of the targets is not current. The first line of the entry specifies the target files and the files on which they depend; the line begins with the target file names, followed by a colon, followed by one or more spaces or tabs, followed by the names of the prerequisite files. It's important to place spaces or tabs after the colon that separates target and dependent files; on systems that allow colons in file names, this allows *make* to distinguish

between the two uses of the colon character.

The commands are on the following lines of the dependency information entry. The first character of a command line must be a tab; *make* assumes that the command lines end with the last line not beginning with a tab.

For example,consider the following dependency entry:

```
prog: prog.o sub1.o sub2.o
        ln -o prog prog.o sub1.o sub2.o -lc
```

This entry says that the file *prog* depends on the files *prog.o*, *sub1.o*, and *sub2.o*. It also says that if *prog* is not current, *make* should execute the *ln* command. *make* considers *prog* to be current if it exists and if it has been modified more recently than *prog.o*, *sub1.o*, and *sub2.o*.

The above entry describes only the dependence of *prog* on *prog.o*, *sub1.o*, and *sub2.o*. It doesn't define the files on which the '.o' files depend. For that, we need either additional dependency entries in the makefile or a rule that can be applied to create '.o' files from '.c' files.

For now, we'll add dependency entries in the makefile for *prog.o*, *sub1.o*, and *sub2.o*, which will define the files on which the object modules depend and the commands to be executed when an object module is not current. In section 1.3 we'll then modify the makefile to make use of *make*'s built-in rule for creating a '.o' file from a '.c' file.

Suppose that the '.o' files are created from the C source files *prog.c*, *sub1.c*, and *sub2.c*; that *sub1.c* and *sub2.c* contain a statement to include the file *defs.h* and that *prog.c* doesn't contain any #include statements. Then the following long-winded makefile could be used to explicitly define all the information needed to make *prog*

```
prog: prog.o sub1.o sub2.o
        ln -o prog prog.o sub1.o sub2.o -lc

prog.o: prog.c
        cc prog.c

sub1.o: sub1.c defs.h
        cc sub1.c

sub2.o: sub2.c defs.h
        cc sub2.c
```

This makefile contains four dependency entries: for *prog*, *prog.o*, *sub1.o*, and *sub2.o*. Each entry defines the files on which its target file depends and the commands to be executed when its target isn't current. The order of the dependency entries in the makefile is not important.

We can use this makefile to make any of the four target files defined in it. If none of the target files exists, then entering

```
make prog
```

will cause *make* to compile and assemble all three object modules from their C source files, and then create *prog* by linking the object modules together.

Suppose that you create *prog* and then modify *sub1.c*. Then telling *make* to make *prog* will cause *make* to compile and assemble just *sub1.c*, and then recreate *prog*.

If you then modify *defs.h*, and then tell *make* to make *prog*, *make* will compile and assemble *sub1.c* and *sub2.c*, and then recreate *prog*.

You can tell *make* to make any file defined as a target in a dependency entry. Thus, if you want to make *sub2.o* current, you could enter

```
make sub2.o
```

A makefile can contain dependency entries for unrelated files. For example, the following dependency entries can be added to the above makefile:

```
hello: hello.o
        ln hello.o -lc

hello.o: hello.c
        cc hello.c
```

With these dependency entries, you can tell *make* to make *hello* and *hello.o*, in addition to *prog* and its object files.

1.3 Rules

You can see that the makefile describing a program built from many .o files would be huge if it had to explicitly state that each .o file depends on its .c source file and is made current by compiling its source file.

This is where rules are useful. When a rule can be applied to a file that *make* has been told to make or that is a direct or indirect prerequisite of it, the rule allows *make* to infer, without being explicitly told, the name of a file on which the target file depends and/or the commands that must be executed to make it current. This in turn allows makefiles to be very compact, just specifying information that *make* can't infer by the application of a rule.

Some rules are built into *make*; you can define others in a makefile. In the rest of this section, we're going to describe the properties of rules and how you write makefiles that make use of *make*'s built-in rule for creating a .o file from a .c file. For more information on rules, including a complete list of built-in rules and how to define rules in a makefile, see section 2.2.

1.3.1 make's use of rules

A rule specifies a target extension, source extension, and sequence of commands. Given a file that *make* wants to make, it searches the rules known to it for one that meets the following conditions:

* The rule's target extension is the same as the file's extension;
* A file exists that has the same basic name as the file *make* is working on and that has the rule's source extension.

If a rule is found that meets these conditions, *make* applies the first such rule to the file it's working on, as follows:

* The file having the source extension is defined to be a prerequisite of the file with the target extension;
* If the file having the target extension doesn't have a command sequence associated with it, the rule's commands are defined to be the ones that will make the file current.

One rule built into *make*, for converting .c files into .o files, says

> "a file having extension '.o' depends on the file having the same basic name, with extension '.c'. To make current such a .o file, execute the command
>
> cc x.c
>
> where 'x' is the name of the file"

Another built-in rule exists for converting .asm files into .o files, using the Manx assembler.

1.3.2 An example

The .c to .o rule allows us to abbreviate the long-winded makefile given in section 1.2 as follows:

```
prog: prog.o sub1.o sub2.o
        ln -o prog prog.o sub1.o sub2.o -lc

sub1.o sub2.o: defs.h
```

In this abbreviated makefile, a dependency entry for *prog.o* isn't needed; using the built-in '.c to .o' rule, *make* infers that the *prog.o* depends on *prog.c* and that the command *cc prog.c* will make *prog.o* current.

The abbreviated makefile says that both *sub1.o* and *sub2.o* depend on *defs.h*. It doesn't say that they also depend on *sub1.c* and *sub2.c*, respectively, or that the compiler must be run to make them current; *make* infers this information from the .c to .o rule. The only information given in the dependency entry is that which *make* couldn't infer by itself: that the two object files depend on *defs.h*.

1.3.3 Interaction of rules and dependency entries

As we showed in the above example, a rule allows you to leave some dependency information unspecified in a makefile. The *prog.o* entry in the long-winded makefile of section 1.2 was not needed, since its information could be inferred by the .c to .o rule. And the dependence of *sub1.o* and *sub2.o* on their respective C source files, and the commands needed to create the object files was also not needed, since the information could be inferred from the .c to .o rule.

There are occasions when you don't want a rule to be applied; in this case, information specified in a dependency entry will override that which would be inferred from a rule. For example, the following dependency entry in a makefile

```
add.o:
    cc -DFLOAT add.c
```

will cause *add.o* to be compiled using the specified command rather than the command specified by the .c to .o rule. *make* still infers the dependence of *add.o* on *add.c*, using the .c to .o rule, however.

2. Advanced features

In the last section we presented the basic features of *make*, with which you can start using *make*. In this section, we present the rest of *make*'s features.

2.1 Dependent Files

A dependent file can be in a different volume or directory than its target file, with the following provisos.

If the file name contains a colon (for example, because the file name defines the volume on which the file is located), the colon must be followed by characters other than spaces or tabs, so that *make* can distinguish between this use of the colon character and its use as a separator between the target and dependent files in a dependency line. This shouldn't be a problem, since most systems don't allow file names to contain spaces or tabs.

All references to a file must use the same name. For example, if a file is referred to in one place using the name

```
/root/src/foo.c
```

then all references to the file must use this exact same name.

2.2 Macros

make has a simple macro capability that allows character strings to be associated with a macro name and to be represented in the makefile by the name. In the following paragraphs, we're first going to describe how to use macros within a makefile, then how they are defined, and

finally some special features of macros.

2.2.1 Using macros

Within a makefile, a macro is invoked by preceding its name with a dollar sign; macro names longer than one character must be parenthesized. For example, the following are valid macro invocations:

```
$(CFLAGS)
$2
$(X)
$X
```

The last two invocations are identical.

When *make* encounters a macro invocation in a dependency line or command line of a makefile, it replaces it with the character string associated with the macro. For example, suppose that the macro OBJECTS is associated with the string *a.o b.o c.o d.o*. Then the dependency entries:

```
prog: prog.o a.o b.o c.o d.o
        ln prog.o a.o b.o c.o d.o

a.o b.o c.o d.o: defs.h
```

within a makefile could be abbreviated as:

```
prog: prog.o $(OBJECTS)
    ln prog.o $(OBJECTS)

$(OBJECTS): defs.h
```

There are three special macros: $$, $*, and $@. $$ represents the dollar sign. The other two are discussed below.

2.2.2 Defining macros in a makefile

A macro is defined in a makefile by a line consisting of the macro name, followed by the character '=', followed by the character string to be associated with the macro.

For example, the macro OBJECTS, used above, could be defined in the makefile by the line

```
OBJECTS = a.o b.o c.o d.o
```

A makefile can contain any number of macro definition entries. A macro definition must appear in the makefile before the lines in which it is used.

2.2.3 Defining macros in a command line

A macro can be defined in the command line that starts *make*. The syntax for a command line definition has the following form:

```
mac=str
```

where *mac* is the name of the macro, and *str* is its value.

str cannot contain spaces or tabs.

For example, the following command assigns the value *-DFLOAT* to the macro *CFLAGS*:

```
make CFLAGS=-DFLOAT
```

The assignment of a value to a macro in a command line overrides an assignment in a makefile statement.

2.2.4 Macros used by built-in rules

make has two macros, CFLAGS and AFLAGS, that are used by the built-in rules. These macros by default are assigned the null string. This can be overriden by a macro definition entry in the makefile.

For example, the following would cause CFLAGS to be assigned the string "-T":

```
CFLAGS = -T
```

These macros are discussed below in the description of built-in rules.

2.2.5 Special macros

Before issuing any command, two special macros are set: $@ is assigned the full name of the target file to be made, and $* is the name of the target file, without its extension. Unlike other macros, these can only be used in command lines, not in dependency lines.

For example, suppose that the files *x.c*, *y.c*, and *z.c* need to be compiled using the option "-DFLOAT". The following dependency entry could be used:

```
x.o y.o z.o:
    cc -DFLOAT $*.c
```

When *make* decides that *x.o* needs to be recreated from *x.c*, it will assign $* the string "x", and the command

```
cc -DFLOAT x.c
```

will be executed. Similarly, when *y.o* or *z.o* is made, the command *cc -DFLOAT y.c* or *cc -DFLOAT z.c* will be executed.

The special macros can also be used in command lines associated with rules. In fact, the $@ macro is primarily used by rules. We'll discuss this more in the description of rules, below.

2.3 Rules

In section 1, we presented the basic features of rules: what they are and how they are used. We also noted that rules could be defined in

the makefile and that some rules are built into *make*. In the following paragraphs, we describe how rules are defined in a makefile and list the built-in rules.

2.3.1 Rule definition

A rule consists of a source extension, target extension, and command list. In a makefile, an entry defining a rule consists of a line defining the two extensions, followed by lines containing the commands.

The line defining the extensions consists of the source extension, immediately followed by the target extension, followed by a colon.

All command lines associated with a rule must begin with a tab character. The first line following the extension line that doesn't begin with a tab terminates the commands for the rule.

For example, the following rule defines how to create a file having extension *.rel* from one having extension *.c*:

```
.c.rel:
    cc -o $@ $*.c
```

The first line declares that the rule's source and target extension are *.c* and *.rel*, respectively.

The second line, which must begin with a tab, is the command to be executed when a *.rel* file is to be created using the rule.

Note the existence of the special macros $@ and $* in the command line. Before the command is executed to create a *.rel* target file using the rule, the macro $@ is replaced by the full name of the target file, and the macro $* by the name of the target, less its extension.

Thus, if *make* decides that the file *x.rel* needs to be created using this rule, it will issue the command

```
cc -o x.rel x.c
```

If a rule defined in a makefile has the same source and target extensions as a built-in rule, the commands associated with the makefile version of the rule replace those of the built-in version. For example, the built-in rule for creating a *.o* file from a *.c* file looks like this:

```
.c.o:
    cc $(CFLAGS) $*.c
```

If you want the rule to generate an assembly language listing, include the following rule in your makefile:

```
.c.o:
    cc $(CFLAGS) -a $*.c
    as -ZAP -l $*.asm
```

2.3.2 Built-in rules

The following rules are built into *make*. The order of the rules is important, since *make* searches the list beginning with the first one, and applies the first applicable rule that it finds.

```
.c.o:
    cc $(CFLAGS) -o $@ $*.c

.asm.o:
    as $(AFLAGS) -o $@ $*.asm
```

The two macros CFLAGS and AFLAGS that are used in the built-in rules are built into *make*, having the null character string as their values. To have *make* use other options when applying one of the built-in rules, you can define the macro in the makefile.

For example, if you want the options -T and -DDEBUG to be used when *make* applies the *.c.o* rule, you can include the line

```
CFLAGS = -T -DDEBUG
```

in the makefile. Another way to accomplish the same result is to redefine the *.c.o* rule in the makefile; this, however, would use more lines in the makefile than the macro redefinition.

2.4 Commands

In this section we want to discuss the execution of operating system commands by *make*.

2.4.1 Allowed commands

A command line in a dependency entry or rule within a makefile can specify any command that you can enter at the keyboard. This includes batch commands, commands built into the operating system, and commands that cause a program to be loaded and executed from a disk file.

2.4.2 Logging commands and aborting make

Normally, before *make* executes a command, it writes the command to its standard output device; and when the command terminates, *make* halts if the command's return code was non-zero. Either or both of these actions can be suppressed for a command, by preceding the command in the makefile with a special character:

@	Tells *make* not to log the command;
-	Tells *make* to ignore the command's return code.

For example, consider the following dependency entry in a makefile:

```
prog: a.o b.o c.o d.o
    ln -o prog a.o b.o c.o d.o -lc
    @echo all done
```

When the *echo* command is executed, the command itself won't be logged to the console.

2.4.3 Long command lines

Makefile commands that start a Manx program, such as *cc*, *as*, or *ln*, or that start a program created with *cc*, *as*, *ln*, and *c.lib*, can specify a command line containing up to 2048 characters.

For example, if a program depends on fifty modules, you could associate them with the macro OBJECTS in the makefile, and also include the dependency entry

```
prog: $(OBJECTS)
    ln -o prog $(OBJECTS) -lc
```

This will result in a very long command line being passed to *ln*.

In the next section we will describe how OBJECTS could be defined.

2.5 Makefile syntax

We've already presented most of the syntax of a makefile; that is, how to define rules, macros, and dependencies. In this section we want to present two features of the makefile syntax not presented elsewhere: comments and line continuation.

2.5.1 Comments

make assumes that any line in a makefile whose first character is '#' is a comment, and ignores it. For example:

```
#
# the following rule generates an 8080 object module
# from a C source file:
#
.c.o80:
    cc80 -o cc.tmp $*.c
    as80 -ZAP -o $*.o80 cc.tmp
```

2.5.2 Line continuation

Many of the items in a makefile must be on a single line: a macro definition, the file dependency information in a dependency entry, and a command that *make* is to execute must each be on a single line.

You can tell *make* that several makefile lines should be considered to be a single line by terminating each of the lines, except the last,

with the backslash character, '\'. When *make* sees this, it replaces the current line's backslash and newline, and the next line's leading blanks and tabs by a single blank, thus effectively joining the lines together.

The maximum length of a makefile line after joining continued lines is 2048 characters.

For example, the following macro definition equates OBJ to a string consisting of all the specified object module names.

```
OBJ = printf.o fprintf.o format.o\
    scanf.o fscanf.o scan.o\
    getchar.o getc.o
```

As another example, the following dependency entry defines the dependence of *driver.lib* on several object modules, and specifies the command for making *driver.lib*:

```
driver.lib: driver.o printer.o \
    in.o \
    out.o
        libutil -o  driver.lib driver.o\
          printer.o \
          in.o out.o
```

This second example could have been more cleanly expressed using a macro:

```
DRIVOBJ= driver.o printer.o\
    in.o out.o

driver.lib: $(DRIVOBJ)
    libutil -o  driver.lib $(DRIVOBJ)
```

This was done to show that dependency lines and command lines can be continued, too.

2.6 Starting make

You've already seen how *make* is told to make a single file. Entering

```
make filename
```

makes the file named *filename*, which must be described by a dependency entry in the makefile. And entering

```
make
```

makes the first file listed as a target file in the first dependency entry in the makefile.

In both of these cases, *make* assumes the makefile is named 'makefile' and that it's in the current directory on the default drive.

In this section we want to describe the other features available when starting *make*.

2.6.1 The command line

The complete syntax of the command line that starts *make* is:

make [-n] [-f makefile] [-a] [-dmacro=str] [file1] [file2] ...

Square brackets indicate that the enclosed parameter is optional.

The parameters *file1*, *file2* ... are the names of the files to be made. Each file must be described in a dependency entry in the makefile. They are made in the order listed on the command line.

The other command line parameters are options, and can be entered in upper or lower case. Their meanings are:

-n	Suppresses command execution. *make* logs the commands it would execute to its standard output device, but doesn't execute them.
-f makefile	Specifies the name of the makefile
-a	Forces *make* to make all files upon which the specified target files directly or indirectly depend, and to make the target files, even those that it considers current.
-dMACRO=str	Creates a macro named MACRO, and assigns *str* as its value.

2.6.2 make's standard output

2.7 Executing commands

When *make* decides that a command needs to be executed, it executes it immediately, and waits for the command to finish. It activates a command whose code is contained in a disk file by issuing an *fexec* function call. It activates DOS built-in commands and batch commands by calling the *system* function, which causes a new copy of the command processor to be loaded. Thus, to use *make*, your system must have enough memory for DOS, *make*, and whatever programs are loaded by *make* to be in memory simultaneously.

2.8 Differences between the Manx and UNIX 'make' programs

The Manx *make* supports a subset of the features of the UNIX *make*. The following comments present features of the UNIX *make* that aren't supported by the Manx *make*.

* The UNIX *make* will let you make a file that isn't defined as a target in a makefile dependency entry, so long as a rule can be applied to create it. The Manx *make* doesn't allow this. For example, if you want to create the file *hello.o* from the file *hello.c* you could say, on UNIX

make hello.o

even if *hello.o* wasn't defined to be a target in a makefile dependency entry. With the Manx *make*, you would have to have a dependency entry in a makefile that defines *hello.o* as a target.

* The UNIX *make* supports the following options, which aren't supported by the Manx *make*:

  p, i, k, s, r, b, e, m, t, d, q

  The Manx *make* supports the option '-a', which isn't supported by the UNIX *make*.

* The special names .DEFAULT, .PRECIOUS, .SILENT, and .IGNORE are supported only by the UNIX *make*.

* Only the UNIX *make* allows the makefile to be read from *make*'s standard input.

* Only the UNIX *make* supports the special macros $<, $?, and $%, and allows an upper case D or F to be appended to the special macros, which thus modifies the meaning of the macro.

* Only the UNIX *make* requires that the suffixes for additional rules be defined in a .SUFFIXES statement.

* Only the UNIX *make* allows macros to be defined on the command line that activates *make*.

* Only the UNIX *make* allows a target to depend on a member of a library or archive.

3. Examples

3.1 First example

This example shows a makefile for making several programs. Note the entry for *arc*. This doesn't result in the generation of a file called *arc*; it's just used so that we can generate *arcv* and *mkarcv* by entering *make arc*.

```
#
# rules:
#
.c.o80:
        cc80 -DTINY -o $@ $*.c
#
# macros:
#
OBJ=make.o parse.o scandir.o dumptree.o rules.o command.o
#
# dependency entry for making make:
#

make: $(OBJ) cntlc.o envcopy.o
        ln -o make $(OBJ) envcopy.o cntlc.o -lc
#
# dependency entries for making arcv & mkarcv:
#
arc: mkarcv arcv
        @echo done

mkarcv: mkarcv.o
        ln -o mkarcv mkarcv.o -lc
arcv : arcv.o
        ln -o arcv  arcv.o -lc
#
# dependency entries for making CP/M-80 versions of arcv & mkarcv:
#
mkarcv80.com: mkarcv.o80
        ln80 -o mkarcv80.com mkarcv.o80 -lt -lc
arcv80.com: arcv.o80
        ln80 -o arcv80.com arcv.o80 -lt -lc

$(OBJ): libc.h make.h
```

3.2 Second example

This example uses *make* to make a library, my.lib. Three directories are involved: the directory *libc* and two of its subdirectories, *sys* and *misc*. The C and assembly language source files are in the two subdirectories. There are makefiles in each of the three directories, and this example makes use of all of them. With the current directory being *libc*, you enter

```
make my.lib
```

This starts *make*, which reads the makefile in the libc directory. *make* will change the current directory to *sys* and then start another *make* program.

This second *make* compiles and assembles all the source files in the *sys* directory, using the makefile that's in the *sys* directory.

When the 'sys' *make* finishes, the 'libc' *make* regains control, and then starts yet another *make*, which compiles and assembles all the source files in the *misc* subdirectory, using the makefile that's in the *misc* directory.

When the 'misc' *make* is done, the 'libc' *make* regains control and builds *my.lib*. You can then remove the object files in the subdirectories by entering

```
make clean
```

3.2.1 The makefile in the 'libc' directory

```
my.lib: sys.mk misc.mk
        rm  my.lib
        libutil -o my.lib -f my.bld
        @echo my.lib done

sys.mk:
        cd sys
        make
        cd ..

misc.mk:
        cd misc
        make
        cd ..

clean:
        cd sys
        make clean
        cd ..
        cd misc
        make clean
        cd ..
```

3.2.2 Makefile for the 'sys' directory

```
REL=asctime.o bdos.o begin.o chmod.o croot.o csread.o ctime.o \
    dostime.o dup.o exec.o execl.o execlp.o execv.o execvp.o \
    fexec.o fexecl.o fexecv.o ftime.o getcwd.o getenv.o  \
    isatty.o localtim.o mkdir.o open.o stat.o system.o time.o\
    utime.o wait.o dioctl.o ttyio.o access.o syserr.o

COPT=
HEADER=../header

.c.o:
        cc $(COPT) -I$(HEADER) $*.c -o $@
        sqz $@
.asm.o:
        as $*.asm -o $@
        sqz $@

all: $(REL)
        @echo sys done

clean:
        rm  *.o
```

3.2.3 Makefile for the 'misc' directory

```
REL=atoi.o atol.o calloc.o ctype.o format.o malloc.o qsort.o \
        sprintf.o sscanf.o fformat.o fscan.o

COPT=
HEADER=../header

.c.o:
        cc $(COPT) -I$(HEADER) $*.c -o $@
        sqz $@
.asm.o:
        as $*.asm -o $@
        sqz $@

all: $(REL)
        @echo misc done

fformat.o: format.c
        cc -I$(HEADER) -DFLOAT format.c -o fformat.o

fscan.o: scan.c
        cc -I$(HEADER) -DFLOAT scan.c -o fscan.o

clean:
        rm  *.o
```

NAME

obd - list object code

SYNOPSIS

obd <objfile>

DESCRIPTION

obd lists the loader items in an object file. It has a single parameter, which is the name of the object file.

NAME

ord - sort object module list

SYNOPSIS

ord [-v] [infile [outfile]]

DESCRIPTION

ord sorts a list of object file names. A library of the object modules that is generated from the sorted list by the object module librarian, *lb*, will have a minimum number of 'backward references'; that is, global symbols that are defined in one module and referenced in a later module.

Since the specification of a library to the linker causes it to search the library just once, a library having no backward references need be specified just once when linking a program, and a library having backward references may need to be specified multiple times.

infile is the name of a file containing an unordered list of file names. These files contain the object modules that are to be put into a library. If *infile* isn't specified, this list is read from *ord*'s standard input. The file names can be separated by space, tab, or newline characters.

outfile is the name of the file to which the sorted list is written. If it's not specified, the list is written to *ord*'s standard output. *outfile* can only be specified if *infile* is also specified.

The *-v* option causes *ord* to be verbose, sending messages to its standard error device as it proceeds.

NAME

set - environment variable utility

SYNOPSIS

set

set VAR=string

DESCRIPTION

set is used to examine and assign values to environment variables.

The first form listed above for the *set* command causes *set* to display the name and value of each environment variable.

The second form assigns *string* to the environment variable *VAR*.

In the second form, if *string* is not given, the specified variable is deleted from the environment.

SEE ALSO

The Tutorial chapter has more information on environment variables.

NAME

setdate - set date and time

SYNOPSIS

setdate

DESCRIPTION

setdate allows you to set the date and time. It prompts you as follows:

```
Date (MM/DD/YY)?
Time (HH:MM:SS)?
```

If you type 02/03, it leaves the year alone. Likewise, if you type 12 to time, it assumes 0 minutes and seconds.

NAME

z - A text editor

SYNOPSIS

z [files]

DESCRIPTION

Z is a text editor which is especially useful for creating source programs in the C programming language. It has the following features:

* It's very similar to the Unix editor *vi*: if you know *vi*, you know *Z*.
* It's a full-screen editor: the screen acts as a window into the file being edited.
* Z has a wealth of commands, and commands are specified with just a few keystrokes, allowing editing to be performed quickly and efficiently. The simple and natural way of entering commands and the mnemonic assignment of commands to keys makes the commands easy to remember and use.
* Z has commands for the following:
    + Bringing different sections of a file into view;
    + Inserting text;
    + Making changes to text;
    + Rearranging text by moving blocks of text around and by inserting text from other files;
    + Accessing files;
    + Searching for character strings and "regular expressions".
* Z has several commands which are useful for editing C programs: there are commands for finding matching parentheses, square brackets, and curly braces; for finding the beginning of the next or preceding function; and for finding the next or preceding blank line.
* Most commands can be easily executed repeatedly.
* Sequences of commands, called macros, can be defined and executed one or more times.
* Changes are made to an in-memory copy of a file; the file itself isn't changed until a command is explicitly given;
* Z has a feature which is useful when editing a large number of related files: the operator can request that a file containing a certain function be edited; Z will automatically find the file and prepare it for editing.

Requirements

Z runs on several systems, including

* IBM PC, running PCDOS version 2.0 or later
* 8086-based systems running CP/M-86 and using an ADM-3A or LSI terminal;
* The Macintosh
* The Amiga
* TRS-80, model 4, using TRSDOS

For 8086-based systems, *Z* requires at least 128KB of memory and allows you to edit programs containing up to 58 K bytes of text.

For 8080- and Z80-based systems, *Z* requires 64 KB of memory and allows you edit programs containing up to 11 K bytes of text.

Components

The *Z* package contains two programs:

Z, the text editor;

ctags, a utility for creating a file which relates tags to C source files.

Preview

The remainder of this description of Z is divided into the following sections:

getting started , which describes how to quickly start using *Z*;

commands and features , which presents an overview of the features and commands of *Z*;

summary , which summarizes the *Z* commands.

1. GETTING STARTED

Z is a very powerful tool for creating and editing C source programs, but its wealth of commands and options can be overwhelming to someone not familiar with it. The purpose of this chapter is to get you using *Z* as quickly as possible, by presenting a small subset of the *Z* commands, with which programs can be created and edited. Then, with the ability to create and edit programs, you can continue reading the rest of this manual at your leisure to learn about the other features and commands of *Z*.

This section is divided into two subsections: the first describes how to create a new C program, and the second how to edit an existing program.

1.1 Creating a new program

Z is activated by entering a command of the form:

```
z hello.c
```

where *hello.c* is the name of the file to be edited. Since we're creating a new program, the file doesn't exist yet, so *Z* says so by displaying a message on its status line (which may be either the first or last line of the display, depending on the system on which Z is running). On systems that use the first display line for status information, the screen then looks like this:

```
"hello.c" no such file or directory
~
~
...
~
~
```

with the cursor on the left-hand column of the second line. On systems that use the last display line for status information, the screen looks like this:

```
~
~
...
~
hello.c doesn't exist
```

with the cursor on the left-hand column of the first line.

Z is now waiting for you to enter a command.

The screen

As mentioned above, *Z* uses the one line of the display for displaying information and for echoing the characters of some commands which are entered. On the Macintosh, the last line is the status line; on other systems, the first line is the status line.

The rest of the lines on the screen are used to display text of the file being edited.

The tilde characters on the screen lines are *Z*'s way of saying that the end of the file has been reached: these characters are not actually in the file.

Modes of Z

Z has two modes: command and insert, which allow you to enter commands and to insert text, respectively.

In this section, we'll spend most of our time in insert mode, using commands only to enter insert mode and to exit *Z*. When we get to the next section, in which we edit a file, we'll discuss more commands.

Insert mode

With *Z* in insert mode, characters that you type are entered into a memory-resident buffer; the characters don't appear in the file until you exit insert mode and explicitly issue a command which causes *Z* to write the buffer to the file.

Z has several commands for entering insert mode; the one we want to use is *i*, which allows text to be entered before the cursor. So type *i*. Notice that *Z* doesn't echo this command on the screen; it only does that for a few commands. Notice also that we are in command mode, as evidenced by the message

<insert mode>

on the right-hand side of the status line.

Now you can enter a program, just as you would on a typewriter. Notice that the cursor is positioned where the next character will be entered. Try entering the "hello world" program:

```
main()
{
   printf("hello, world\n");
}
```

When you hit the <return> key after entering the printf line, the cursor was left positioned on the next line of the screen underneath the first non-white space character of the preceding line. This feature, which is known as "autoindent", is useful when creating C programs, encouraging statements within a compound statement to be indented and lined up. Autoindent can be disabled and enabled, and we'll show you how later.

We want the closing curly brace of the main function to be on the first column of the line, not indented. So type the backspace key to get back to the first column, and then type the '}' key.

The backspace key can also be used to backspace over characters that you incorrectly type.

When you're done inserting the program, hit the escape key to exit insert mode and return to command mode. The key used as the escape key varies from system to system. On the IBM PC, the key labeled ESC is the escape key. On the TRS-80, models III and 4, the key labeled BREAK is the escape key. And on the Macintosh, the backquote key, `, is the escape key.

Exiting Z

To write the program you've just entered from the text buffer to the disk file *hello.c* and then exit Z, type *ZZ*.

Occasionally you may want to exit Z without writing the text you've entered to a file; in this case, type

:q!

followed by a carriage return, CR.

1.2 Editing an Existing File

In this section we're going to present a few commands which will allow you to make changes to an existing file.

Starting and stopping Z

You get in and out of *Z* when editing an existing file just as you do when creating a new file. To start *Z*, enter

```
z hello.c
```

where hello.c is the name of the file to be edited. And to stop *Z* and save the changes you've made, put *Z* in command mode and enter:

```
ZZ
```

Z knows if you made changes to the original text or not; if you did, it saves the original file by changing the extension of its name to *.bak* and then writes the modified text to a new file having the specified name. If a *.bak* file with that name already exists, it will be deleted before the rename occurs.

If you didn't make any changes, the *ZZ* command causes *Z* to halt without changing any disk files.

The command *:q!* will cause *Z* to halt without writing anything to the file being edited.

Going back to the startup of *Z*, *Z* reads the specified file into the text buffer, displays the first screenful of the file's text, displays the file's statistics (name, number of lines, number of characters) on the status line, positions the cursor at the first character of the first line, and enters command mode, waiting for you to enter a command.

The cursor

Before describing the commands for viewing and changing the text in *Z*'s memory-resident buffer, we need to discuss the cursor.

In *Z*, the character position in the text which is pointed at by the cursor acts as a reference point: most commands perform an action relative to that position. For example, the *i* command, described in the last section, allows you to enter text before the cursor. And the *x* command, to be discussed, deletes the character at which the cursor is located.

So we will be describing two types of commands in this section: those that move the cursor around in the text, thus bringing different sections of text into view, and those that modify text in the vicinity of the cursor.

Moving around in the text: scrolling

The text you created in section 1, for the "hello, world" program, easily fit on a single screen. But most text files are too large to be

viewed all at once, so we need commands to bring different sections into view.

Two such commands are the "scroll" commands: "scroll down", represented by the character control-D, and "scroll up", represented by control-U. That is, to execute the "scroll down" command, you hold down the control key and then depress the 'D' key.

The key used as the control key differs from system to system. On the IBM PC, it's the key labeled 'Ctrl'. On the Macintosh, it's the cloverleaf key (the key next to the 'Option' key that has the unusual symbol).

In the rest of this manual, we will refer to control characters using notation of the form ^D rather than control-D, for brevity. Thus, the "scroll up" and "scroll down" commands are represented as ^U and ^D, respectively.

A scroll command moves the screen up or down in the file, bringing another half-screen's worth of text into view. It's as if the text was on a reel of tape and the screen is a viewer: scrolling down moves the viewer down the reel, and scrolling up moves the viewer up the reel.

When scrolling, the cursor will be left on the same position within the text after the scroll as before, if that position is still within view. Otherwise, the cursor is moved to a line in the text which was newly brought into view.

Moving around in the text: the 'Go' command

Scrolling is one way to move around in the text, but it's slow. If we have a large text file, to which we want to append text, it would take a long time and many scroll commands to reach the end.

The *go* command, *g*, is one way to move rapidly to the point of interest in the text: entering *g* by itself will move the cursor to the end of the text and, if necessary, redraw the screen with the text which precedes it.

The *g* command can also be preceded by the number of the line of interest; in this case, the cursor is moved to the beginning of that line. So to move back to the first line of text, enter:

```
1g
```

The *g* command can be used to move to any line within the text, but since you usually don't know the numbers of the lines, the *g* command is mainly used to move to the beginning and end of the text.

Moving around in the text: string searching

So, scrolling allows us to take a casual stroll through text, and the *g* command to move rapidly to the beginning and end of the file. What

we need is a command to rapidly move to a specific point in the middle of the text.

The "string search" command, /, is such a command. When you enter /, followed by the string of interest, followed by a carriage return, *Z* searches forward in the text from the cursor position, looking for the string If *Z* reaches the end of the text without finding the string, it will "wrap around", and continue searching from the beginning of the text.

If the string is found, the cursor is positioned at its first character and, if necessary, the screen is redrawn with its surrounding text.

If the string isn't found, a message saying so is displayed on the status line of the screen and the cursor isn't moved.

While the "string search" command and its string are being entered, the characters are displayed on the status line, and normal editing operations can be used, such as backspacing over mistyped characters.

Z remembers the last string searched for. To repeat the search, enter the "find next string" command, *n*.

Finely tuned moves

With the commands presented up to now, you can move to the area of interest in the text. The next few paragraphs present commands which move the cursor from somewhere within the area of interest to a specific character position, from which changes will be made.

Some commands for this, from the many available in *Z*, are:

- and *CR* (carriage return)
  Move the cursor up and down one line, respectively, to the first non-whitespace character on the line;

space and backspace
  Move the cursor right and left, respectively, on the line on which the cursor is located.

These commands can be preceded by a number, which cause the command to be performed the specified number of times. For example,

```
3-
```

moves the cursor up three lines, and

```
5<space>
```

moves the cursor right five characters. Note that <space> represents the space bar.

Deleting text

You now have a repetoire of commands which allow you to move the cursor fairly quickly to any location in a text file. We're ready to

move on to a few commands for modifying the text.

Two such commands, for deleting text, are "delete character", *x*, and "delete line", *dd*:

x Deletes the character under the cursor;

dd Deletes the entire line on which the cursor is located.

Each of these commands can be preceded by a number, causing the command to be repeated the specified number of times. For example,

2x

deletes two characters, and

3dd

deletes three lines.

More insert commands

You already know one command for inserting text: *i*, which allows text to be inserted before the cursor. We need a few more insert commands:

a Enters insert mode such that text is inserted following the cursor;

o Creates a blank line below the current line (ie, the line on which the cursor is located), moves the cursor to the new line, and enters insert mode;

O Same as *o*, but the new line is above the current line.

Summary

With the set of commands presented in this chapter, you can edit any text file. You should continue reading this manual, to learn more about *Z*, while you use the basic command set for performing your editing chores.

You'll find that *Z* has many more capabilities, which allow you to perform functions more quickly, with fewer keystrokes, than with the basic command set, and which allow you to perform functions which you can't perform with the basic command set.

The commands in the basic set are listed on the next page.

Starting and stopping Z

Z filename	Start Z, and prepare 'filename' for editing
ZZ	Stop Z, and write modified text to the edit file
:q!	Stop Z, without writing anything to the edit file

scrolling

^D	Move down half a screenful
^U	Move up half a screenful

Moving the cursor

g	Move the cursor to the end of the text, or to a specific line
/str	Search for the character string "str" and move the cursor to it
n	Search again, using the same string
-	Move cursor up a line
CR	Move cursor down a line
space	Move cursor right one character
backspace	Move cursor left one character

Inserting text

i	Insert before cursor
a	Insert after cursor
o	Insert new line below current line
O	Insert new line above current line

Deleting text

x	Delete character under cursor
dd	Delete line on which cursor is located

2. More commands

In this section we're going to describe the rest of the features and commands of Z, building and expanding on the information presented in the previous chapter. The section is organized into subsections; some describe a group of related commands, some a particular feature, and some how to perform a specific function with Z.

2.1 Introduction

Before getting into the *Z* commands, we want to discuss in more detail the way that *Z* displays information on the screen and the way that commands are entered.

2.1.1 The screen

We've already discussed the basic details on *Z*'s use of the screen. There's just a few more things to discuss: the display of unprintable characters and the display of lines which don't fit on the screen.

2.1.1.1 Displaying unprintable characters

A file edited by *Z* can contain any character whose ASCII value in decimal is less than 128, including unprintable characters, such as SOH, LF, and ESC. *Z* displays unprintable characters as two characters; the first is ^, and the second is the character whose ASCII value equals that of the character itself plus 0x40. For example, the unprintable character SOH is displayed as the pair of characters ^*A*, since the ASCII value of SOH is 1, and 1 plus 0x40 is 0x41, which is the ASCII value for the character 'A'.

2.1.1.2 Displaying lines that don't fit on the screen

In the previous chapter we said that lines beyond the end of the file are displayed with the character ~ in the first column of the line on the screen. When you see the ~ character in the leftmost column of a line on the screen, this usually signifies that this line of the display doesn't contain a line of text. Lines which don't fit on the screen are displayed by *Z* in a similar manner, as you'll soon see.

Z allows lines to be entered which are longer than a screen line. Normally, *Z* simply displays such lines on several screen lines. In some cases, however, the entire line won't fit on the screen. For example, if the cursor is positioned at the beginning of the file, it may not be possible to display the text of an entire big line at the bottom of the screen. In this case, *Z* displays an @ character in the first column of the screen lines on which the text would be displayed.

Thus, when you see the @ character in the leftmost column of a line on the screen, this usually signifies that the text which would have appeared on this line of the screen was too big, and not that the @ character is in the text.

2.1.2 Commands

When most commands are entered, *Z* doesn't echo the characters on the screen. For some commands, however, it does. In this latter category are the commands whose first character begins with *:* and with the string search commands.

For these commands, the characters are displayed on the screen's status line, and can be backspaced over and reentered, if necessary.

Also, Z doesn't act on such commands until you type the carriage return key, CR.

2.1.3 Special Keys

There are two keys that have special meaning for Z: the escape key, which is used to exit insert mode, and the control key, which is used in conjunction with another key to generate control characters. The actual keys used for these functions varies from system to system, as mentioned in the previous chapter.

The escape key is ESC on the IBM PC. On the TRS-80, models III and 4, it's the BREAK key. And on the Macintosh, it's the backquote key, `.

The control key is 'Ctrl' on the IBM PC. On theMacintosh it's the key next to the 'Option' key that has the cloverleaf symbol.

On the Macintosh, there are times when you want to generate a backquote, and not escape. For example, backquote is a cursor motion command to Z. To generate backquote, hold down the control key (the key next to the option key), and then type backquote.

2.2 Paging and Scrolling

In the last chapter we described commands for scrolling through text, ^*U* and ^*D*. Another pair of commands allow you to page, instead of scroll, through text. They are ^*B* and ^*F*, which page backwards and forwards, respectively.

A page command brings the previous or next screenful of text into view by redrawing the screen with the new text. Whereas scrolling was described as a viewer moving over a reel of tape, paging can be described as the turning of pages of a book.

Paging moves you through text more quickly than scrolling does. However, since paging redraws the screen all at once, while scrolling changes it gradually, it's often more difficult to keep a sense of continuity when paging than when scrolling. As an aid to continuity when paging, two lines of text which were previously in view are still in view after paging.

In the discussion of scrolling in the last chapter, we neglected to mention that the scroll commands can be preceded by a value specifying the number of lines to be scrolled up or down. If a number isn't specified, the last scroll value entered is used; if a scroll value was never entered, it defaults to half a screen's worth of lines. Separate values are maintained for scrolling up and for scrolling down.

The scrolling and paging commands necessarily move the cursor within the text, but they can't be used to home the cursor to an exact position at which changes are to be made. For this, you'll have to use commands described in subsequent sections.

2.3 Searching for strings

In the previous chapter, we described the string search command, /, which causes Z to scan forward, looking for the string. In this section, we describe the rest of the searching capabilities of Z. First, the rest of the string searching commands are described; then, the capability of Z to match patterns called "regular expressions", of which specific character strings are a special case, is described.

2.3.1 The other string-searching commands

The other string-searching commands are:

? Behaves like /, but Z finds the previous occurence of the string rather than the next;

n Repeats the last string-search command;

N Repeats the last string-search command, but in the opposite direction;

:se ws=0 and :se ws=1
Turns the wrap scan option off or on, respectively.

When Z reaches the end or beginning of text without finding the string of interest, it normally "wraps around" to the opposite end of the text and continues the search. It does this because by default the "wrap scan" option is on. This option can be disabled by entering the "set option" command:

:se ws=0

thus causing the search to end when it reaches the end of text. The option can be reenabled by entering:

:se ws=1

Note that for this colon command, as for all colon commands, carriage return must be typed before the command is executed.

2.3.2 Regular expressions

The string you tell Z to search for is actually a "regular expression". A regular expression is a pattern which is matched to character strings. The pattern can define a specific sequence of characters which comprise the string; in this case, only that specific string matches the pattern. The pattern can also contain special characters which match a class of characters; in this case, the pattern can match any of a number of character strings.

For example, one such special construct is square brackets surrounding a character string; this matches any character in the enclosed string. So the regular expression

ab[xyz]cd

matches the strings

abxcd
abycd
abzcd

Another special character is *, which matches any number of occurences of the preceding pattern. For example, the regular expression

ab*c

matches many strings, including

ac
abc
abbc

and so on. And the pattern

ab[xyz]*cd

matches many strings, including:

abcd
abxcd
abxycd
abzzxcd

and so on.

The complete list of special characters and constructs that can be included in regular expressions is:

^	When the first character of a pattern, it matches the beginning of the line
$	When the last character of a pattern, it matches the end of the line;
.	Matches any single character;
<	Matches the beginning of a word;
>	Matches the end of a word;
[str]	Matches any single character in the enclosed string;
[^str]	Matches any single character *not* in the enclosed string;
[x-y]	Matches any character between x and y;
*	Matches any number of occurences of the preceding pattern.

2.3.3 Disabling extended pattern matching

The "magic" option enables and disables the extended pattern matching capability. To turn off this option, enter:

:se ma=0

And to turn it on, enter:

:se ma=1

By default, extended pattern matching is disabled.

With the magic option off, only the characters ^ and $ are special in patterns.

2.4 Local Moves

In this section we're going to present more commands for moving the cursor fairly short distances; up or down a few lines, along the line on which it's located, and so on. We've already presented several, namely CR (carriage return), space, and backspace; but there are many more, reflecting the importance of finely-tuned, quickly-executed movements to the user.

2.4.1 Moving around on the screen:

Here are some commands for moving the cursor short distances:

h	Moves to the left one character;
j	Moves down one line, leaving the cursor in the same column;
k	Moves up one line, leaving the cursor in the same column;
l	Moves right one cursor;

The keys ^H, LF, ^K, and ^L are synonyms for h,j,k, and l, respectively.

These commands can be preceded by a number, which specifies the number of times the command is to be repeated.

Z has commands for moving the cursor to the top, middle, and bottom of the screen; they are H, M, and L, respectively. The cursor is positioned at the beginning of the line to which it's moved.

Remember the - command, which moved the cursor up a line, to the first non-whitespace character? As you might expect, + will move the cursor down a line, to the first non-whitespace character. + is thus equivalent to CR, the command presented in the last chapter.

2.4.2 Moving within a line

We've already presented several commands for moving the cursor around within the line on which it's located:

h, ^H, backspace	Left one character;
l, ^L, space	Right one character;

Here are a few more:

^	Moves the cursor to the first non-whitespace character on the line;
0	Moves to the first character on the line;
$	Moves to the last character on the line;

A few commands fetch another character from the keyboard, search for that character, beginning at the current cursor location, and leave the cursor near the character:

f	Scan forward, looking for the character, and leave the

cursor on it;

t Same as *f*, but leave the cursor on the character preceding the found character;

F Same as f, but scan backwards;

T Same as t, but scan backwards.

; Repeat the last f, t, F, or T command;

, Repeat the last f, t, F, or T command in the opposite direction.

Finally, the command | moves the cursor to the column whose number precedes the command. For example, the following command moves the cursor to column 56 on the current line:

```
56|
```

2.4.3 Word movements

Z has several commands for moving the cursor to the beginning or end of a word which is near the cursor:

w Moves to the beginning of the next word;

b Moves to the beginning of the previous word;

e Moves to the end of the current word.

For the preceding commands, a "word" is defined in the normal way: a string of alphabetical and numerical characters surrounded by whitespace or punctuation. There is a variant of each of these commands, differing only in the definition of a "word": they think that a word is any string of non-whitespace characters surrounded by whitespace. The variant of each of these commands is identified by the same letter, but in upper case instead of lower:

W Moves to the beginning of the next big word;

B Moves to the beginning of the previous big word;

E To the end of the current big word.

Each of these commands can be preceded by a number, specifying the number of times the command is to be repeated. For example,

```
5w
```

moves forward five words.

The word movement commands will cross line boundaries, if necessary, to find the word they're looking for.

2.4.4 Moves within C programs

Z has several commands for moving the cursor within C programs:

]] and [[Move to the opening curly brace, {, of the next or previous function, respectively;

% Move to the parenthesis, square bracket, or curly bracket which matches the one on which the cursor is currently located;

{ and } Move to the preceding or next blank line.

The [[and]] commands assume that the opening and closing curly braces for a function are in the first column of a line, and that all other curly braces are indented.

As an example of the '%' command, given the statement:

```
while (((a = getchar()) != EOF) && (c != 'a'))
```

with the cursor on the parenthesis immediately following the 'while', the % command will move the cursor to the last closing parenthesis on the line.

2.4.5 Marking and returning

Z has commands which allow you to set markers in the text and later return to a marker. Twenty six markers are available, identified by the alphabetical letters.

Unlike the other commands described in this section, these commands are not limited to moves within the current area of the cursor - they can move the cursor anywhere within the text.

A marker is set at the current cursor location using the command

*m*x

where x is the letter with which you want to mark the location.

There are two commands for returning to a marked position:

`x	Moves the cursor to the location marked with the letter 'x';
'x	Moves the cursor to the first non-whitespace character on the line containing the 'x' marker.

Remember, to generate backquote on the Macintosh, hold down the control key and then type backquote.

Occasionally, you may accidently move the cursor far from the desired position. There are two single quote commands for returning you to the area from which you moved:

``	Returns the cursor to its exact starting point;
''	Returns the cursor to the first non-whitespace character on the line from which the cursor was moved.

For example, if the cursor is on the line:

```
if (a >= 'm' && a <= 'z')
```

at the character '<', then following a command which moves the cursor far away, the command `` will return the cursor to the '<' character, and the command '' will return it to the beginning of the word 'if'.

2.4.6 Adjusting the screen

The *z* command is used to redraw the screen, with a certain line at the top, middle, or bottom of the screen.

To use it, place the cursor on the desired line, then enter the *Z* command, followed by one of these characters:

CR	To place the line at the top of the screen;
.	To place it in the middle of the screen;
-	To place it at the bottom.

The *z* command isn't a true cursor motion command, because the cursor is in the same position in the text after the command as before.

2.5 Making changes

That concludes the presentation of cursor movement commands. The next several sections describe commands for making changes to the text.

2.5.1 Small changes

In this section we present commands for making small changes. We've already presented two such commands in the previous chapter:

x	Which deletes the character at which the cursor is located;
dd	Which deletes the line at which the cursor is located.

The other commands are:

X	Delete the character which precedes the cursor; can be preceded by a count of the number of characters to be deleted;
D	Delete the rest of the line, starting at the cursor position;
rx	Replace the character at the cursor with 'x';
R	Start overlaying characters, beginning at the cursor. Type the escape key to terminate the command. (Remember, this key differs from system to system);
s	Delete the character at the cursor and enter insert mode; when preceded by a number, that number of characters are deleted before entering insert mode;
S	Delete the line at the cursor and enter insert mode; when preceded by a count, that number of lines are deleted before entering insert mode;
C	Delete the rest of the line, beginning at the cursor, and enter insert mode;
J	Join the line on which the cursor is positioned with the following line; when preceded by a count, that many lines are joined.

2.5.2 Operators for deleting and changing text

Z has a small number of commands, called 'operators', for modifying text. They all have the same form, consisting of a single letter command, optionally preceded by a count and always followed by a cursor motion command. The count specifies the number of times the command is to be executed. The command affects the text from the current cursor position to the destination of the cursor motion command, if the starting and ending position of the cursor are on the same line. If these positions are on different lines, the command affects all lines between and including the lines which contain the starting position and ending positions.

In this section, we're going to describe the operators for deleting and changing text, *d* and *c*:

d	Deletes text as defined by the cursor motion command;
c	Same as *d*, but *Z* enters insert mode following the deletion.

For example,

dw	Deletes text from the current cursor location to the beginning of the next word;
3dw	Deletes text from the cursor to the beginning of the third word;
d3w	Same as '3dw';
db	Deletes text from the current to the beginning of the previous word;
d`a	Deletes text from the cursor to the marker 'a', if the marker and the starting cursor position are on the same line. Otherwise, deletes lines from that on which the cursor is located through that on which the marker is located; On the Macintosh, generate backquote by holding down the control key and then typing the backquote key.
d/var	Deletes text either from the cursor to the string "var" or between the lines at which the cursor is currently located and that on which the string is located.
d$	Deletes the rest of the characters on the line, and hence is equivalent to *D*.

2.5.3 Deleting and changing lines

In the last chapter, we presented a command for deleting lines: *dd*. As you can see now, this is a special form of the *d* command, because the character following the first *d* is not a cursor motion command.

For all the operator commands, typing the command character twice will affect whole lines. Thus, typing *cc* will clear the line on which the cursor is located and enter insert mode. Preceding *cc* with a number will compress the specified number of lines to a single blank line and enter insert mode on that line.

2.5.4 Moving blocks of text

When text is deleted using the *d* or *c* command, it's moved to a buffer called the "unnamed buffer". (There are other buffers available, which have names. More about them later).

Data in the unnamed buffer can be copied into the main text buffer using one of the "put" commands:

p	Copies the unnamed buffer into the main text buffer, after the cursor;

P Same as *p*, but the text is placed before the cursor.

Thus, the delete and put commands together provide a convenient way to move blocks of text within a file.

The contents of the unnamed buffer is very volatile: when any command is issued that modifies the text, the text which was modified is placed in the unnamed buffer. This is done so that the modification can be 'undone', if necessary, using one of the 'undo' commands. For example, if you delete a character using the *x* command, the deleted character is placed in the unnamed buffer, replacing whatever was in there. So you have to be careful when moving text via the unnamed buffer: if you delete text into the unnamed buffer, expecting to put it back somewhere, then issue another command which modifies the text before issuing the put command, the deleted text is no longer in the unnamed buffer.

As you'll see, the named buffers can also be used to move blocks of text, and their contents are not volatile.

2.5.5 Duplicating blocks of text: the 'yank' operator

The 'yank' operator, *y*, copies text into the unnamed buffer without first deleting it from the main text buffer. When used with the 'put' command, it thus provides a convenient way for duplicating a block of text.

Since *y* is an operator, it has the same form as the other operators: an optional count, followed by the *y* command, followed by a cursor motion command. The command yanks the text from the cursor position to the destination of the cursor motion command, if the starting and ending positions are on the same line. If they are on separate lines, a whole number of lines are yanked, from that on which the cursor is currently located through that to which the cursor would be moved by the cursor motion command. The text is yanked into the unnamed buffer.

For example,

yw	Copies text from the cursor to the next word into the unnamed buffer;
y3w	Copies text from the cursor to the beginning of the third word;
3yw	Same as 'y3w';
y'a	Copies text from the cursor location to the marker 'a' into the unnamed buffer, if the two positions are on the same line. Otherwise, copies entire lines between and including those containing the two positions.

As a special case, the command *yy* will yank a specified number of whole lines. The command *Y* is a synonym for *yy*. For example,

yy Yanks the line at which the cursor is located;

3Y Yanks three lines, beginning with the one on which the cursor is located.

2.5.6 Named buffers

In addition to the unnamed buffers, *Z* has twenty six named buffers, each identified by a letter of the alphabet, which can used for rearranging text. Text can be deleted or yanked into a named buffer and put from it back into the main text buffer.

The advantage of these buffers over the unnamed buffer in rearranging text is that their contents are not volatile: when you put something in a named buffer, it stays there, and won't be overwritten unexpectedly. Also, as you'll see, the named buffers can be used to move text from one file to another.

To yank text into a named buffer, use the yank operator, preceded by a double quote and the buffer name, and followed by a cursor motion command. For example, the following will yank three words into the 'a' buffer:

```
"ay3w
```

and the following yanks four lines into the 'b' buffer, beginning with the line on which the cursor is located:

```
"b4yy
```

Text is deleted into a named buffer in the same way: the delete command is used, preceded by a double quote and the buffer name. For example, to delete characters from the cursor to the 'a' marker into the 'h' buffer:

```
"hd`a
```

The preceding command, when the source and destination cursor positions are on separate lines, will delete a number of whole lines into the 'h' buffer, from that on which the cursor is initially located through that containing the destination position.

As you remember, on the Macintosh, the backquote key is interpreted as the escape key. To generate a real backquote for use in the preceding example, you must hold down the control key (the key with the strange symbol next to the option key) and then type backquote.

To delete ten lines into the 'c' buffer:

```
"c10dd
```

Text in a named buffer is put back into the main text using the 'put' commands *p* and *P*, preceded by a double quote and the buffer name. For example:

ap Puts text from the 'a' buffer, after the cursor;

zP Puts text from the 'z' buffer, before the cursor.

2.5.7 Moving text between files

The named buffers are conveniently used to move text from one file to another. First yank or delete text from one file into a named buffer; then switch and begin editing the target file, using the *:e* command:

:e filename

(More on this later). Then move the cursor to the desired position; then put text from the named buffer.

This technique only works when using named buffers, not with the unnamed buffer. When switching to a new file, the unnamed buffer is cleared, but the named buffers are not.

2.5.8 Shifting text

The last two operator commands to introduce are the 'shift' operators, < and >, which are used to shift text left and right a tabwidth, respectively.

For example,

>/str

shifts right one tab width the lines from that on which the cursor is located through that containing the string "str".

Following the standard operator syntax, repeating the shift operator twice affects a number of whole lines:

5<< Shifts five lines left;
>> Shifts one line right.

2.5.9 Undoing and redoing changes

Z remembers the last change you made, and has a command, *u*, which undoes it, restoring the text to its original state.

Z also remembers all the changes which were made to the last line which was modified. Another 'undo' command, *U*, undoes all changes made to that line.

Finally, the period command, '.', reexecutes the last command that modified text.

2.6 Inserting text

We've already presented most of the commands for entering insert mode:

a	Append after cursor;
i	Insert before cursor;
o	Open new line below cursor;
O	Open new line above cursor;
C	Delete to end of line, then enter insert mode;
s	Delete characters, then enter insert mode;
S	Delete lines, then enter insert mode;

In this section we want to present the remaining few commands for entering insert mode, and present some other features of insert mode.

2.6.1 Additional insert commands

The other commands for entering insert mode are:

A	Append characters at the end of the line on which the cursor is located. This is equivalent to *$a*;
I	Insert before the first non-whitespace character on the current line. This is equivalent to *^i*.

2.6.2 Insert mode commands

Some editing can be done on text entered during insert mode, using the following control characters:

backspace	Delete the last character entered;
^H	Same as 'backspace' character;
^D	Same as "backspace";
^X	Erase to beginning of insert on current line;
^V	Enter next character into text without attempting to interpret it.

^V is used to enter non-printing characters into the text. For example, to enter the character 'control-A' into the text, type

^V^A

That is, hold down the control key, then type the 'V' key, then the 'A' key, then release the control key. As mentioned earlier, non-printing characters are displayed as two characters: '^' followed by a character whose ASCII code equals that of the non- printing character plus 0x40.

2.6.3 Autoindent

The *Z* 'autoindent' option is useful when entering C programs. When you are in insert mode and type the 'carriage return' key, with the autoindent option enabled, the cursor will be automatically indented on the new line to the same column on which the first non-whitespace character appeared on the previous line. This feature is useful for editing C programs because it encourages statements which

are part of the same compound statement to be indented the same amount, thus making the program more readable.

Z autoindents a line by inserting tab and space characters at the beginning of a new line. If you don't want to be indented that much, you can backspace over these automatically inserted tabs and spaces until you reach the desired degree of indentation.

The autoindent option can be selectively enabled and disabled using the 'set options' command:

:se ai=0 to disable autoindent
:se ai=1 to enable autoindent

When *Z* is activated, autoindent is enabled.

2.7 Macros

Z allows you to define a sequence of commands, called a 'macro', and then execute the macro one or more times.

When a macro is defined to *Z*, it's placed in a special buffer, called the macro buffer, and then executed once. There are two ways to define a macro to *Z*: immediately and indirectly.

2.7.1 Immediate macro definition

An 'immediate' macro definition is initiated by typing the characters

```
:>
```

Z responds by clearing the status line, displaying these characters on the line, and waiting for you to enter the sequence of commands.

As you enter the commands, *Z* displays them on the status line and enters them immediately into the macro buffer; that's why it's called 'immediate macro definition'.

If you make a mistake while entering commands, you can simply backspace and enter the correct characters.

To terminate the definition, type the carriage return key. *Z* will then execute the sequence of commands in the macro buffer. The contents of this buffer are not altered by executing the macro, so you can reexecute the macro without reentering it, as described below.

2.7.2 Some examples

The following macro advances the cursor one line, and deletes the first word on the new line:

```
+dw
```

contains two commands: +, which advances the cursor, and *dw*, which deletes the word beneath the cursor.

The next macro moves the cursor to the previous line and deletes the last character on the line:

```
-$x
```

It contains three commands: -, which moves the cursor to the previous line; *$*, which moves the cursor to the last character on that line; and *x*, which deletes the character beneath the cursor.

You can also insert text using a macro. You enter insert mode using one of the normal insert commands. The characters which follow the insert command on the macro line, up to a terminating escape character, are then inserted into the text. The escape character causes *Z* to return to command mode and continue executing commands in the macro which follow the insert command.

Remember, the key used as the escape character differs from system to system. See section 1 of this chapter for details.

For example, the following macro advances the cursor to the next line, deletes the second word on the line, inserts the character string "and furthermore", and deletes the last word on the line:

```
+wdwiand furthermore<ESC>$bdw
```

The last macro contains the following commands:

+	Advances the cursor to the next line;
w	Moves the cursor to the second word on the line;
dw	Deletes the word beneath the cursor;
iand furthermore<ESC>	Inserts the text "and furthermore". <ESC> stands for the escape key;
$	Moves the cursor to the last character on the line;
b	Moves the cursor to the beginning of the last word on the line;
dw	Deletes that word.

Z also allows you to search for a string from within a macro. Enter in the macro the 'string search' command (for example, /), followed by the string, followed by the ESC character. For example, the following macro moves the cursor to the word "Ralph" and deletes it:

```
/Ralph<ESC>dw
```

It contains the commands

/Ralph<ESC>	Moves the cursor to "Ralph". <ESC> stands for the escape key;
dw	Deletes "Ralph".

The following macro finds "Ralph" and replaces it with "Sarah":

```
/Ralph<ESC>cwSarah<ESC>
```

It contains the commands:

/Ralph<ESC>	Moves the cursor to "Ralph";
cwSarah<ESC>	Changes "Ralph" to "Sarah".

2.7.3 Indirect macro definition

The other way of defining a macro is to yank a line containing a sequence of commands from the main text buffer into a named buffer and then have *Z* move the contents of the named buffer to the macro buffer.

Commands for indirect macro definition are:

@x	Causes *Z* to move the contents of the 'x' buffer to the macro buffer and then execute it once;
xv	A synonym for '@x'.

Indirect macro definition of macros has several advantages over immediate definition: for one, if a macro defined immediately is incorrect, you have to reenter the entire macro. With an indirectly defined macro, you can edit the macro definition in the main text buffer and then move it back to the macro buffer.

Another advantage is that you can store several macros in the named buffers and easily reexecute a macro, without having to reenter it. With immediate definition, when a new macro is defined, the previously defined macro is lost, and must be reentered to be reexecuted.

One difference between entering macros immediately and via the text buffer and named buffer concerns the method for specifying the end of a search string and for exiting insert mode. With immediate definition, you do this by typing the ESC key directly. For indirect definition, in which the macro is first entered into the main text buffer, typing the ESC key would cause *Z* to exit insert mode, not to enter the ESC key into to text of the macro. In this case, you enter the ESC key by first typing control-V, then ESC. This causes *Z* to enter the ESC character into the text of the macro and remain in insert mode.

2.7.4 Re-executing macros

Once a macro is defined and is in the macro buffer, it can be re-executed by typing one of the commands:

@@
v

Preceding the command with a count will cause the macro to be executed the specified number of times.

2.7.5 Wrapping around during macro execution

While executing a macro, *Z* may reach the beginning or end of the text, and want to continue beyond that point. This is especially true when reexecuting macros. The 'macro wrap' option, *wm*, specifies whether *Z* should terminate the macro execution at that point, or continue at the opposite end of the text.

This option is enabled and disabled using the 'set options' command:

:se wm=0	To disable macro wrapping;
:se wm=1	To enable it.

When *Z* starts, this option is enabled.

2.8 The Ex-like commands

The 'substitute' and 'repeat last substitution' commands are part of a set of commands that are being added to the Z editor and that are similar to commands in the UNIX Ex editor. In this section we will first generally describe the syntax of these commands, then the 'substitute' command, and finally the 'repeat last substitution' command.

The Ex-like commands consist of a leading colon, followed by zero, one, or two addresses identifying the lines to be affected by the command, followed by a single-letter command, followed by command parameters, and terminated by a carriage return. Most commands have a default set of lines that they affect, thus frequently allowing you to enter commands without explicitly specifying a range.

These commands support regular expressions, as defined in the Z documentation, for identifying addresses and strings to be searched for.

2.8.1 Addresses in Ex commands

An address can be one of the following:

* A period, ., addresses the current line; that is, the line on which the cursor is located.
* The character $ addresses the last line in the edit buffer.
* A decimal number *n* addresses the *n*-th line in the edit buffer.
* 'x addreses the line marked with the mark name *x*. Lines are marked with the *m* command.
* A regular expression surrounded by slashes (/) addresses the first line containing a string that matches the regular expression. The search begins with the line following the current line and continues towards the end of the edit buffer. If a line isn't found when the end of the buffer is reached, and if Z's *ws* option is set to 1 (ie, by the *:se ws=1* command) the search continues at the beginning of the buffer, stopping when the current line is reached.
* A regular expression surrounded by question marks (?) also addresses the first line containing a string that matches the regular expression. But in this case, the search begins with the line preceding the current line in the edit buffer and continues towards the beginning of the buffer. If a line isn't found when the beginning of the buffer is reached, and if Z's *ws* option is set to 1 (ie, by the *:se ws=1* command) the search continues at the end of the buffer, stopping when the current line is reached.
* An address followed by a plus or a minus sign, which in turn is followed by a decimal number *n* addresses the *n*-th line

following or preceding the line identified by the address.

When two addresses are entered to define the range of lines affected by a command, the addresses are usually separated by a comma. They can also be separated by a semicolon; in this latter case, the current line is set to the line defined by the first address, and then the line corresponding to the second address is located.

When no value is specified for the first address in an address range, it's assumed to be the current line or the first line in the buffer, depending on whether the second address was preceded with a comma or a semicolon. When no value is specified for the second address in an address range, it's assumed to be the last line in the buffer. Thus, if neither the beginning nor the ending address of a range is specified, the range consists of either all the lines in the buffer or the lines from the current through the last line in the buffer, depending on whether comma or semicolon is used to separate the unspecified addresses.

2.8.2 The 'substitute' command

The 'substitute' command has the following form:

:[range]s/pat/rep/[options]

where square brackets surround a parameter to indicate that the parameter is optional.

Z searches the lines specified by *range* for strings that match the regular expression *pat*, replacing them with the *rep* string. If *range* isn't specified, just the current line is searched. When the command is completed, the cursor is left on the character following the last replaced string.

Normally, Z automatically replaces a string that matches *pat*. Specifying *c* as an *option* causes Z instead to pause when it finds a matching string, ask if you want the string to be replaced, and make the replacement only if you give your permission.

Normally, Z will replace only the first *pat*-matching string on a line. Specifying *g* as an option causes Z instead to replace all matching strings on a line; in this case, after Z replaces a string on a line, it continues searching for more strings on the line at the character following the replaced string.

An ampersand (&) in the replacement string *rep* is replaced by the string that matched *pat*. The special meaning of *&* can be suppressed by preceding it with a backslash, \.

A replacement string consisting of just the percent character (%) is replaced in the current substitution by the replacement string that was used in the last substitution. The special meaning of % can be suppressed by preceding it with a backslash, \.

2.8.2.1 Examples

:s/abc/def/

Search the line on which the cursor is located for the string *abc*; if found, replace it with the string *def*.

:1,$s/ab*c/xyz/

Search all lines in the edit buffer for strings that begin with *a*, end in *c*, and have zero or more *b*'s in between; replace such strings with *xyz*. On any given line, only the first occurrence of a string that matches the pattern is replaced.

:/{/;/}/s/for/while/c

Find the first line following the current line that contains a {; then find the first line following this line that contains a }. In the lines between and including these lines, search for the string *for*; for each such string, ask if it should be replaced; if yes, replace it with *while*.

2.8.3 The '&' (repeat last substitution) command

The & command has the form

:[range]&

where brackets indicate that the parameters are optional.

The & command causes the last 'substitute' command to be executed again, using the same search pattern, replacement string, and options as were used in the previous command. The command searches the lines that are specified in the & command's *range*; if *range* isn't specified, the substitution is performed on just the current line.

2.9 Starting and stopping Z

You already know how to start and stop *Z*, from the previous chapter. In this section we present more information related to the starting and stopping of *Z*.

2.9.1 Starting Z

In the previous chapter, we said that *Z* was started by specifying the name of the file to be edited on the command line:

```
Z filename
```

Z can also be started without specifying a file name or by specifying a list of files to be edited.

2.9.1.1 Starting Z without a filename

When *Z* is started without a filename being specified, you will normally tell *Z* the name of the file to be edited, once it's active, using the *:e* command:

```
:e filename
```

It isn't absolutely necessary for *Z* to know the name of the file you're editing: *Z* will allow you to create and modify text in the text buffer without knowing the name of the file to which you intend to write the text. But then you'll have to explicitly tell *Z* to write the text, using the command

```
:w filename
```

Z can't automatically write the text, since it doesn't know which file you're editing.

2.9.1.2 Starting Z with a list of files

Z can be started and passed a list of names of files to be edited, as follows:

```
Z file1 file2 ...
```

Z will remember the list, and make the first file in the list the 'edit file'; that is, read the file into the main text buffer and allow it to be edited.

Z has a command, *:n*, which will make the next file in the list the edit file, after writing the contents of the text buffer back to the current edit file.

File lists are discussed in more detail below.

2.9.1.3 The options file

Z has several options for controlling its operation in different situations. You've already met most of them, including the 'autoindent', 'macro wrap', and other options. The complete list of

options will be presented later. In this section, we want to present another feature of Z related to options; the ability to set options automatically, when Z is started.

When Z starts, it will read options from the file named 'z.opt', if it exists. Z looks for the file in different places on different systems.

On PCDOS and on the Macintosh, the environment variable ZOPT defines the name of the options file. If this variable doesn't exist, or if the file isn't found there, Z then looks for the file *z.opt* on the current directory on the default drive.

Each line in the options file defines the value of one option, with a statement of the form

```
opt=val
```

where 'opt' is the name of the option, and 'val' is its value. For example, the following sets the 'tab width' option to 8 characters:

```
ts=8
```

2.9.1.4 Setting options for a file

When Z makes a file the 'edit file' by reading it into the edit buffer, the file itself can specify the options to be in effect during its edit session. This feature is most useful in editing files which have different tab settings.

A file specifies option values by including strings of the form

```
:opt=val
```

in the first ten lines of the file. For example, the following line could be used near the front of a C program, causing a tab width of 8 characters to be used:

```
/*  :ts=8    */
```

When Z starts editing a file, the tab width is set back to the default

value, 4 characters, before the file is scanned for option settings.

2.9.2 Stopping Z

In the preceding chapter we presented the following commands for stopping *Z*:

ZZ	If the file's text in the edit buffer has been modified, the text is written to the file, after changing the extension of the original file to ".bak".
:q!	Stops *Z* without writing the text to the file.

Two other commands for exiting *Z* are:

:wq	Which is the name as ZZ, except that the text in the main text buffer is always written to the file, even if no changes have been made;
:q	Which conditionally stops *Z*. If no changes were made to the file's text, *Z* stops; otherwise, it displays a message and remains active.

2.10 Accessing files

Z has other commands for accessing files besides ZZ and :wq, and we're going to discuss them in this chapter.

Z usually knows the name of the file you are editing, and in the sections that follow we will call this the 'edit file'. *Z* makes use of this knowledge, allowing you to write to the edit file without specifying it by name. For example, the ZZ command writes text to the edit file without requiring you to enter the name of the file.

Some commands allow you to access files without redefining *Z*'s idea of the edit file. The commands described in the next two subsections fall into this category.

Other commands cause *Z* to terminate editing of one file and begin editing another; this new file becomes the edit file. The commands described in the other sections of this section are of this type.

2.10.1 File names

In the *Z* commands that require a file name, the name is usually entered using the standard system conventions. However, some characters are special to *Z*:

#	Refers to the last edit file;
%	Refers to the current edit file;
\	Causes the next character to be used in the filename and not be interpreted.

To enter a file name which contains these characters, precede the special character with the character '\'. For example, on PCDOS, to edit the file

```
a:\subs\hello.c
```

use the command

```
:e \\subs\\hello.c
```

On PCDOS, the '/' character can also be used as a separator between directories and between a directory and file name. Thus, the above command could also be entered as:

```
:e /subs/hello.c
```

2.10.2 Writing files

The command *:w* writes the contents of the main text buffer to a file, without redefining the identity of the current edit file. It has the following forms:

:w	Write to the current edit file;
:w	Write to the specified file;
:w!	Same as ':w filename', but the file is overwritten if it exists.

As with all colon commands, carriage return must be typed to cause *Z* to execute the command.

When entered without a filename, *:w* creates a new file having the name of the current edit file and writes the contents of the edit buffer to it. This form of the :w command is commonly used to periodically save text during a long edit session, to guard against system failures.

The option *bk* tells *Z* whether it should save the original edit file before creating a new one. If *bk* is 1 the original will be saved, and if 0 it won't. *Z* saves the original file by changing its name to *.bak*. An existing *.bak* file will be erased before the rename occurs. For details on setting options, see the Options section.

When a filename is entered with the :w command, the text is written to that file, if it doesn't already exist. If it does, nothing is written, and *Z* displays a message on the status line; in this case you must use the *:w!* form of the command to overwrite the file.

The *:w!* command unconditionally writes the text to the specified file, after truncating the file, if it exists, so that nothing is in it. Unlike the *:w* command which doesn't specify a file name, the :w! command doesn't save the original file as a ".bak" file.

2.10.3 Reading files

The command

:r filename

merges one file with a file being edited, without redefining the identity of the edit file.

It reads the contents of the specified file into the main text buffer, inserting the new text following the line on which the cursor is located. It doesn't alter text which is already in the edit buffer.

2.10.4 Editing another file

The following commands cause *Z* to stop editing one file and begin editing another, which thus becomes the 'edit file':

:e	Edit the specified file;
:e!	Edit the file, discarding changes to the current edit file;
:e	Reload the current edit file;
:e!	Reload the current edit file, discarding changes;
:e	Re-edit the previous edit file;
^^	Synonym for ':e #'. (the command is 'control-^').

Z begins editing another file by erasing the contents of the main text buffer and the unnamed buffer, resetting the tab width to four characters, redrawing the display with the first screenful of lines from the file, and setting the cursor at the first character in the text.

When switching to a new edit file, Z doesn't change the contents of the named buffers. Thus, these buffers can be used to hold text which is to be moved from one file to another and to contain commonly used macros.

The command

:e filename

causes the specified file to conditionally become the edit file. The condition is that changes must not have been made to the text of the current edit file since it was last written to disk. If this condition is met, then the switch is made; otherwise, Z displays a message on the status line and nothing is changed: the identity of the edit file is the same, the contents of the edit buffer are not modified, and the options are not changed.

If Z doesn't let you switch edit files when you enter

:e filename

and you want to save the changes to the current edit file, enter the sequence:

:w
:e filename

You can unconditionally cause Z to begin editing a new file by entering:

:e! filename

In this case, Z doesn't care whether or not you made changes to the current edit file since it was last written to disk; it begins editing the new file without changing the previous edit file.

Sometimes the text in the edit file may get hopelessly scrambled, and you want to get a fresh copy of the edit file contents. The command

:e!

specified without a file name will do just that.

Z not only remembers the name of the current edit file you're editing; it remembers the name of the last file you edited as well. Z allows you to refer to this name using the character '#' in :e commands, thus providing a quick means to re-edit the previous edit file:

:e #

causes the previous edit file to conditionally become the current edit file, and

```
:e! #
```

causes it to unconditionally become the edit file.

The command ^^ (that is, control-^) is a synonym for ':e #'.

Z also remembers the position at which the cursor was located in the previous edit file, and when you begin re-editing this file it sets the cursor back to this position.

2.10.5 File lists

Z's 'file list' feature is convenient to use when you have several files to edit: you pass *Z* a list of the files and begin editing the first one. When you're finished with one file, a command switches to the next file in the list, after automatically saving the changes to the current edit file. An option to the command prevents *Z* from saving changes, and another command "rewinds" the file list so that you're back editing the first file in the list again.

There's two ways to pass the list of files to be edited to *Z*: as parameters to the command that starts *Z*, and as parameters to the ':n' command. In each case, *Z* remembers the list and makes the first file in the list the 'edit file'. For example,

```
Z file1 file2 file3
```

starts *Z* and defines the list of files file1, file2, and file3. *Z* makes file1 the edit file; that is, prepares it for editing by reading it into the edit buffer and displaying its first lines.

When *Z* is active, the command

```
:n file4 file5 file 6
```

defines a new list of files: file4 file5 and file6. *Z* makes file4 the edit file.

When used without a files list, the ':n' command switches from one file in the list to the next:

:n	Writes the text in the edit buffer to the current edit file before switching;
:n!	Switches without writing anything to the current edit file.

The ':rew' command "rewinds" the file list; that is, makes the first file in the list the edit file. This command behaves like the ':n' command, in that it by default writes changes to the current edit file before rewinding; and when an exclamation mark is appended to the comand, the rewind occurs without writing to the current edit file.

2.10.6 Tags

Z has a feature useful for editing large C programs which contain many functions distributed over several files. With the aid of a cross-

reference file relating 'tags', that is, function names, to the files containing them, you simply tell *Z* the name of the function that you want to edit and *Z* makes the file containing it the edit file by reading it into the edit buffer and positioning the cursor to the function.

The following commands specify the tag of the function to be edited:

:ta tag	Position to the function named 'tag' in the appropriate file, if the current edit file is up to date;
:ta! tag	Same as ':ta tag', but the switch to the new file occurs even if the current edit file isn't up to date.

When using the ':ta' command, the current edit file is considered 'up to date' if the text in the edit buffer hasn't been modified since it was last written to the file. When used without the trailing '!', the ':ta' command won't switch edit files if the current edit file isn't up to date; it'll just display a message on the status line. You can then either write the text in the edit buffer to the file and re-enter the ':ta' command, or immediately enter the ':ta!' command, to switch edit files anyway.

The command

^]

that is, control-], is convenient when, while editing or viewing one function, you want to edit or examine a function which it calls. You just set the cursor to the name of the called function and enter '^]'; *Z* will make the file containing the called function the edit file, and position the cursor to this function.

For example, while examining the file *crtdvr.c*, you may come across a call to the function *pcdvr*, and want to take a look at it. By positioning the cursor at the beginning of the word 'pcdvr' and typing '^]', *Z* will make the file containing *pcdvr* the edit file and leave the cursor positioned at this function.

2.10.7 The CTAGS utility

The utility program *ctags* creates the cross reference file, *tags*, which relates function names to the file containing them.

ctags is activated by a command of the form

ctags file1 file2 ...

where file1, ..., are names of files whose functions are to be placed in the cross reference file. A file name can specify a group of files using the character '*'. For example:

*.c

specifies all files whose extension is ".c", and

```
f*.c
```

specifies all files whose first character is 'f' and whose extension is ".c".

ctags considers a character string in a file it is scanning to be a function name, for inclusion in the cross reference file if it's a valid C name which begins on the first column of a line and which is terminated by an open parenthesis character. Thus, the function which begins

```
FILE *
fopen(...
```

would be included in the cross reference, but the function which begins

```
FILE * fopen(...
```

wouldn't.

ctags creates the cross reference file, *tags*, in the current directory on the default drive.

When a tags command is given, Z searches for this file in locations which differ from system to system. On PCDOS, it searches for the file in the current directory on the default drive.

2.11 Executing system commands

On PCDOS, *Z* has two commands which allow you to execute system commands while *Z* is active and then return to *Z*:

:!cmd	Executes the system command 'cmd';
:!!	Re-executes the last command.

For example,

```
:!dir *.c
```

executes the system command 'dir *.c' and returns to *Z*.

2.12 Options

Z has has several options under user control which define how *Z* behaves in certain situations. Most of these options have been discussed peripherally in previous sections, when appropriate. In this section we want to focus on the options.

Each option is identified by a code. The options and their codes are:

ai — The 'auto-indent' option. When this option is enabled and you begin inserting text on a new line, *Z* automatically indents the line by inserting tabs and spaces so that the first character you type will be located in the same column as the first non-whitespace character on the previous line. By default, this option is enabled.

eb — The 'error bells' option. When this option is enabled, *Z* will beep when you make a mistake. By default, this option is enabled.

ma — The 'magic' option. When this option is enabled, regular expessions used in string searches can include extended pattern matching characters. Otherwise, only the characters '^' and '$' are special and the extended pattern matching constructs are gotten by preceding them with ". By default, this option is disabled.

ts — The 'tab set' option. Specifies the number of characters between tab settings. By default, the tab width is four characters.

wm — The 'wrap on macro' option. When this option is enabled, and a macro being executed reaches the end of the buffer, the macro will wrap around to the beginning of the buffer and continue. By default, this option is enabled.

ws — The 'wrap on search' option. When this option is enabled, and a search for a string reaches the end of the buffer without finding the string, the search continues at the opposite end of the buffer. By default, this option is enabled.

bk — This option defines whether Z, when a *:w* command is entered to write the edit buffer to the current edit file, should save the original edit file before creating a new one.

An option is enabled by setting it to 1, and disabled by setting it to 0.

2.13 Z vs. Vi

Z is very similar to the UNIX editor VI:

* Both are full-screen editors, display text in the same way, and reserve one line of the display for messages;
* They have the same two modes: command and insert;
* Z supports most of the Vi commands. The Z commands are activated by the same keystrokes and perform the same functions as their Vi counterparts.

Z and Vi differ in the following ways:

* In Z, the buffer in which text is edited is entirely within RAM memory; in Vi, the buffer is both in memory and on disk. Because of this, Z is restricted in the size of program that can be edited, but Vi is not;
* A single copy of Vi can be configured to use any type terminal. A single copy of Z is pre-configured to use just one terminal;
* Vi has an underlying editor, *ex*, whose commands can be executed while Vi is active. Z doesn't have an underlying editor. However, Z does support some *ex* commands directly; these are the commands whose first character is ':'. (Vi interprets the ':' as a request to execute the *ex* command which is entered after the ':');
* Vi has commands and options useful for editing documents and for editing LISP programs, but Z doesn't;
* With Vi, you can create a shell and suspend Vi while executing commands from within the new shell. With some Vis, you can also suspend Vi while executing commands from the shell that activated Vi. Z doesn't support either of these features, although it will allow you to suspend Z while executing a single system command;
* Vi saves the last nine deleted blocks of text, and has commands with which it can recover them, if necessary. Z lets you recover the last deleted block;
* With Vi, operator commands can affect exactly the characters between the starting and ending cursor positions, even when the positions are on different lines. It has variations of these commands which allow whole lines to be affected, between and including the lines containing the two positions.

  In Z, operator commands in which the starting and ending cursor positions are on different lines always affect whole lines, between and including the lines containing the two positions.

3. Command Summary

Starting Z

z name	edit file name
z name1 name2	edit file name1, rest via :n

The Display

~ lines	lines past end of file
@ lines	lines that don't fit on screen
^x	control characters
tabs	expand to spaces, cursor on last

Options

ai=1/0	auto-indent on/off
eb=1/0	error bells on/off
ma=0/1	magic off/on
ts=val	tab width (4)
wm=1/0	wrap on search when executing macro
ws=1/0	wrap on search scan
bk=1/0	save original file as *.bak*

Adjusting the Screen

^F	forward screenful
^B	backward screenful
^D	scroll down half screen
^U	scroll up half screen
zCR	redraw, current line at top
z-	redraw, current line at bottom
z.	redraw, current line at center

Positioning within File

g	go to line (default is end of file)
G	go to line (default is end of file)
/pat	move cursor to pat searching forwards
?pat	move cursor to pat searching backwards
n	repeat last / or ?
N	repeat last / or ? in reverse direction
]]	next "^{"
[[	previous "^{"
%	find matching (), {}, or [].

Marking and Returning

`	previous context
'	first non-white at previous context
mx	mark position with letter 'x'
`x	to mark 'x'
'x	first non-white at mark 'x'

Line Positioning

H	top of screen
M	middle of screen
L	bottom of screen
+	next line, first non-white
CR	next line, first non-white
-	previous line, first non-white
LF	next line, same column
j	next line, same column
^K	previous line, same column
k	previous line, same column

Character Positioning

0	beginning of line
^	first non-white at beginning of line
$	end of line
space	forward a character
^L	forward a character
l	forward a character
^H	backwards a character
h	backwards a character
fx	find character 'x' forward
Fx	find character 'x' backwards
tx	position before character 'x' forward
Tx	position before character 'x' backwards
;	repeat last f, F, t or T
,	repeat last f, F, t or T in reverse direction
|	move to specified column number

Words and Paragraphs

w	word forward
W	blank delimited word forward
b	back word
B	back blank delimited word
e	end of word
E	end of blank delimited word
}	to next blank line
{	to previous blank line

Insert and Replace

a	append after cursor
A	append at end of line
i	insert before cursor
I	insert before first non-blank in line
o	open line below current line
O	open line above current line
rx	replace single character with 'x'
R	replace characters

Corrections During Insert

^H	erase last character
^D	erase last character
^X	erase to beginning of insert on current line
^V	insert following character directly

Operators

d	delete
c	delete and insert
<	left shift
>	right shift
y	yank

Miscellaneous Operations

D	delete rest of line
C	change rest of line
s	substitute characters
S	substitute lines
J	join lines
x	delete characters starting at cursor
X	delete characters before cursor
Y	yank lines

Yank and Put

p	put after current
P	put before current
"xp	put from buffer 'x'
"xy	yank to buffer 'x'
"xd	delete to buffer 'x'

Undo and Redo

u	undo last change
U	restore current line
.	repeat last change command

Macros

@x	execute macro in buffer 'x'
"xv	execute macro in buffer 'x'
@@	repeat last macro
v	repeat last macro

Colon Commands

:e name	edit file name
:e	reedit last file
:e! name	edit file name, discarding changes
:e!	reedit last file, discarding changes
:e #	edit alternate file
^^	edit alternate file
:e! #	edit alternate file, discarding changes
:r name	read file "name" into current file
:w	write back to file being edited
:wq	write back to file and quit
:w name	write to file "name" if does not exist
:w! name	write to file "name", delete if exists
:q	quit
:q!	quit, discarding changes
:x	quit, saving file if modified
ZZ	quit, saving file if modified
:f	show current file and line
^G	show current file and line
:n	edit next file in list
:n!	edit next file in list, discarding changes
:n arg1 arg2	specify new list
:rew	point back to beginning of list
:rew!	point back to beginning, discarding changes
:ta tag	position to tag in appropriate file
^]	same as :ta using word at cursor
:ta! tag	position to tag, discarding changes
:!cmd	execute cmd, then return (PCDOS only)
:!!	re-execute last cmd (PCDOS only)
:>macro	specify and execute immediate macro
:set opt1=val opt2=val ...	set editor options
:se opt1=val opt2=val ...	set editor options
:set all	display current option settings
:[range]s/pat/rep/[options]	substitute *rep* for *pat* in *range*
:[range]&	repeat last substitute command

LIBRARY FUNCTIONS OVERVIEW:
AMIGA INFORMATION

Library Functions Overview: Amiga Information

The *Library Functions Overview* chapter presented overview information that is independent of the system on which your programs run. This chapter presents overview information about the library functions that is specific to programs that run on an Amiga.

The sections of this chapter are numbered; the information discussed in a section is related to the section in the *Library Functions Overview* chapter that has the same number.

1. Overview of I/O: Amiga Information

When the UNIX-compatible I/O functions supplied with Aztec C68k are used to access files and devices, a program can have at most ten files and devices open simultaneously. This includes files and devices open for both standard and unbuffered i/o, and the standard i/o devices. When this limit is reached, an open file or device must be closed before another can be opened.

Amiga functions are also supplied for accessing files and devices; a program can access some files and devices using the Amiga functions, and others using the UNIX-compatible functions.

The UNIX-compatible I/O functions contain tables having a fixed number of entries, one for each open file or device. The number of entries in this table sets the limit on the number of files and devices that can be accessed simultaneously using the UNIX-compatible functions. Thus, files being accessed by the Amiga functions don't directly affect the number of files that can be open simultaneously for access by the Amiga functions. However, the UNIX-compatible I/O functions access files and devices using the Amiga functions; so AmigaDOS limits on the number of files and devices that are accessed simultaneously indirectly causes files and devices being accessed by the Amiga functions to affect the number of files and devices that can be accessed simultaneously by the UNIX-compatible functions.

1.1 Pre-opened devices and command line arguments

The standard i/o devices are pre-opened for programs running in the CLI environment, and are redirected as specified on the CLI command line. A program's standard error device is opened to the same file or device as the program's standard output.

Values can be passed to a program running in the CLI environment, as arguments to the program's *main* function (for details on how *main* receives them, see section 1.1 of the chapter entitled "Library Overview: System Independent Information"). When a program is started by the CLI, the arguments specified on the command line are passed to the program; when it is started by another program, the arguments specified in the calling program's *exec* or *fexec* call are passed to it.

For a program started by the Workbench, no devices are preopened, and values can't be passed to its *main* function. The first argument to the program's *main* function is 0, and the second is a pointer to the workbench startup message.

1.2 File I/O

Unlike some systems supported by Aztec C, positioning of a file relative to its end is always correctly done, with the following limitation: you can't position a file beyond its end.

1.2.1 Device I/O

Using the UNIX-compatible I/O functions, a program can access devices, using their AmigaDOS names. For a list of these names, and a description of device I/O using AmigaDOS, see the AmigaDOS manuals.

A program can also access devices using Amiga function calls. These calls bypass the UNIX-compatible I/O system and AmigaDOS, and give a program the most control over devices. For example, when accessing the serial device using the Amiga functions, a program has control over the device's baud rate, etc, which it doesn't when accessing the serial device using the UNIX-compatible or AmigaDOS functions.

The UNIX-compatible functions supplied with Aztec C support console I/O differently on other systems than they do on the Amiga. On other systems, there is one console device, and a program can issue calls to the *ioctl* function to define how console I/O is to be performed. On the Amiga, *ioctl* is not supported; a program accesses the *con:* device for line-oriented console I/O, and the *raw:* device for character-oriented console I/O. To gain the most control over console I/O, a program should use the Amiga functions.

1.2.1.1 Aborting a program from the console

To abort a program, the operator can type ^C (ie, control-C). The functions *open*, *read*, *write* and *lseek* check for ^C by calling the *Chk_Abort* function.

When called, *Chk_Abort* aborts the program if (1) ^C has been typed and (2) the global *short Enable_Abort* is non-zero (which it is by default). If either of these conditions isn't satisfied, *Chk_Abort* simply

returns.

Thus, to disable the ability of the operator to abort a program by typing ^C, a program must set *Enable_Abort* to 0.

5. Overview of Dynamic Buffer Allocation: Amiga Information

Two sets of functions for dynamic buffer allocation are provided with the Amiga: one set is in *c.lib*, while the other is in the object module *heapmem.o*.

The main difference between the two sets of functions is that the *c.lib* functions allocate buffers from whatever memory is available, while the *heapmem.o* functions get a large, contiguous, block of memory when the first call to them is made, and then allocate buffers from it.

Another difference, which results from the first difference, is that there's more buffer allocation functions in *heapmem.o* than in *c.lib*. In particular, *heapmem.o* supports the following functions, which behave as described in the Library Functions chapters:

* *malloc*
* *calloc*
* *realloc*
* *free*
* *sbrk*

c.lib supports *malloc*, *calloc*, and *free*, which also behave as described in the Library Functions chapters, but it doesn't support *realloc* or *sbrk*.

For the *heapmem.o* functions, the default size of the initially-allocated block of memory is 40k bytes. A program can specify a different size for this block by setting the size in the global *long _Heapsize* before calling any buffer allocation or standard i/o functions.

When a program terminates, all blocks of memory that have been allocated by the above-described functions are automatically released.

The memory-allocation functions described above call the Amiga *AllocMem* and *FreeMem* function to get and free memory, and allow the Amiga to make the allocations from any available section of memory. A program requiring more control over the region from which buffers are allocated can itself call the Amiga memory allocation functions, thus bypassing the Aztec functions. However, in this case, the program must explicitly release any such buffers before it terminates.

AZTEC C68K/AMIGA FUNCTIONS

Chapter Contents

Amiga Functions lib68
Index 4
The functions 5

Aztec C68k/Amiga Functions

This chapter describes Aztec functions that are available to programs running on an Amiga but that may not be available to Aztec-compiled programs running on other systems.

This chapter is divided into sections, each of which describes a group of related functions.

As with description of the system independent functions, the header to a section of this chapter has a parenthesised letter that specifies the library containing the section's functions. The codes and their related libraries are:

C	c.lib;
S	s.lib;

Index to the Aztec C68k/Amiga Functions

function	*page*	*description*
access	ACCESS	determine file accessibility
assert	ASSERT	verify program assertion
brk	BREAK	set heap pointer
execl, etc	EXEC	jump to another program
exit	EXIT	terminate program
_exit	EXIT	terminate program
fexec, etc,	FEXEC	call another program
getenv	GETENV	get value of environment variable
mktemp	MKTEMP	make name for temporary file
perror, etc	PERROR	write error message
sbrk	BREAK	set heap pointer
scdir	SCDIR	scan directory for files
scr_curs, etc	SCREEN	screen manipulation
time, etc	TIME	time functions
tmpfile	TMPFILE	create & open temporary file
tmpnam	TMPNAM	make name for temporary file

NAME

access - determine accessibility of a file or directory

SYNOPSIS

```
int access (filename, mode)
char *filename;
int mode;
```

DESCRIPTION

access determines whether a file or directory can be accessed in the way that the calling function wants to access it. It can also be used to just test for the existence of a file or directory.

filename points to the name of the file or directory; this name optionally contains the drive and path of directories that must be passed through to get to the file or directory. If the drive component isn't specified, the file or directory is assumed to reside on the default drive. If the path component isn't specified, the file or directory is assumed to reside in the current directory on the specified drive.

mode is an *int* that specifies the type of access desired:

mode	*meaning*
4	read
2	write
1	execute (if a file) or search (if a directory)
0	check existence of the file or directory.

If the existence of the file or directory is being checked (ie, mode=0), *access* returns 0 if the file exists and -1 if it doesn't. In the latter case, *access* also sets the symbolic value ENOENT in the global integer *errno.*

When *access* is called to determine if a file can be accessed in a certain way (ie, *mode* isn't 0), *access* returns 0 if the file can be accessed in the desired manner; otherwise, it returns -1 and sets a code in the global integer *errno* that defines why the access is not permitted.

When asked, *access* says that a directory can be read or written; this means that a program can create and delete files on the directory, not that it can directly read and write the directory itself.

The symbolic values that *access* may set in *errno* when it's called with a non-zero *mode* parameter are:

errno	*meaning*
ENOTDIR	A component of the path prefix is not a directory.

ENOENT	The file or directory doesn't exist.
EACCES	The file or directory can't be accessed in the desired manner.

SEE ALSO

The "Errors" section of the Library Overview chapter discusses *errno*.

NAME

assert - verify program assertion

SYNOPSIS

#include <assert.h>

assert (expr)
int expr;

DESCRIPTION

assert is useful for putting diagnostic messages in a program. When executed, it will determine whether the expression *expr* is true or false. If false, it prints the message

Assertion failed: *expr*, file *fff*, line *lnnn*

where *fff* is the name of the source file and *nnn* is the line number of the *assert* statement.

To prevent assertion statements from being compiled in a program, compile the program with the option *-DNDEBUG*, or place the statement *#define NDEBUG* ahead of the statement *#include <assert.h>*.

NAME

sbrk - memory allocation function

SYNOPSIS

void *sbrk(size)
unsigned int size;

DESCRIPTION

sbrk, which is contained along with alternate versions of other memory-allocation functions in the file *heapmem.o*, provides an elementary means of allocating and deallocating space from a contiguous block of memory. When first called, *sbrk* gets a block of memory by calling the Amiga function *AllocMem*. The default size of this block is 40K bytes. A program can specify a different size by setting the desired size in the global *long* *_Heapsize* before issuing its first call to a memory-allocation function or to a standard i/o function.

sbrk maintains a pointer to the top of allocated space within the block. When called, *sbrk* increments this pointer by *size* bytes and returns the value that the pointer had on entry.

When a program terminates, the block of memory that was allocated by *sbrk*, if any, is automatically released.

SEE ALSO

For an overview of the memory-allocation functions, see the Overview of Library Functions chapters.

ERRORS

If an *sbrk* request would make the *sbrk* pointer go past the end of *sbrk*'s block of memory, *sbrk* will return -1 as its value, without modifying its pointer.

NAME

execl, execv, execlp, execvp

SYNOPSIS

execl(name, arg0, arg1, arg2, ..., argn, 0)
char *name, *arg0, *arg1, *arg2, ...;

execv(name, argv)
char *name, *argv[];

execlp(name, arg0, arg1, arg2, ..., argn, 0)
char *name, *arg0, *arg1, *arg2, ...;

execvp(name, argv)
char *name, *argv[];

DESCRIPTION

The *exec* functions can be used within the CLI environment to load and start another program. The called program is loaded on top of the calling program; thus, if the *exec* function succeeds, it doesn't return to the caller.

The following paragraphs will first describe the parameters to the *exec* functions, then describe the differences between the functions, and finally discuss other features of the functions.

Parameters

name is the name of the file containing the program to be loaded.

The exec functions can pass arguments to the called program. *execl* and *execlp* build a command line by concatenating the strings pointed at by *arg1*, *arg2*, and so on. If a C program is being called, its *main* function will see *arg0* as *argv[0]*, *arg1* as *argv[1]*, and so on.

execv and *execvp* build a command line by concatenating the strings pointed at by *argv[0]*, *argv[1]*, and so on. The *argv* array must be have a null pointer as its last entry. If a C program is being called, its *main* function will see the calling function's *argv[i]* as its *argv[i]*.

The Functions

execl and *execv* load a program from the specified file: *execl* is useful when a fixed number of arguments are being passed to a program. *execv* is useful for programs which are passed a variable number of arguments.

execlp and *execvp* will search for a program first in the current directory, then in the C: directory.

Other Information

If an exec function fails, for example because the file doesn't exist, *exec* terminates the calling program.

Before another program is called, all files and devices that were opened in the calling program for standard and unbuffered i/o, except for those associated with the standard in, standard out, and standard error units, are closed.

SEE ALSO

The FEXEC functions also load and execute another program; they differ from the EXEC functions in that they return to the caller.

NAME

exit, _exit

SYNOPSIS

exit(code)

_exit(code)

DESCRIPTION

exit and *_exit* terminate the calling program, after releasing all dynamically-allocated memory that was obtained by calling *malloc*, *calloc*, *realloc*, or *sbrk*. The functions differ in that *exit* closes all files opened for standard and unbuffered i/o, while *_exit* doesn't.

If the program was started from within a CLI environment, the program's return code is set to *code*. Control then returns to a waiting program, which is usually the CLI. When the program was activated by another program's *fexec* call, it's this calling program that is waiting and that resumes when the called program exits.

If the program was started from the Workbench, then it was executing as an independent process. In this case, the program's process is deleted and the space associated with its stack, segment list, and process structure is released.

NAME

fexecl, fexecv

SYNOPSIS

fexecl(name, arg0, arg1, arg2,..., argn, 0)
char *name, *arg0, *arg1, *arg2, ..., *argn;

fexecv(name, argv)
char *name, *argv[];

wait()

DESCRIPTION

The *fexec* functions, which can be used by programs running in the CLI environment, load and call another program. The calling program is suspended while the called program is executing; the *fexec* function returns when the called program terminates.

wait returns as its value the return code from the *fexec*-executed program.

The parameters

name specifies the name of the file from which the program is to be loaded, and optionally, the drive on which it's located and the path to it.

The *fexec* functions can pass arguments to the called program. *fexecl* builds a command line by concatenating the strings pointed at by *arg0*, *arg1*, and so on. If a C program is being called, its *main* function will see *arg0* as *argv[0]*, *arg1* as *argv[1]*, and so on.

fexecv builds a command line by concatenating the strings pointed at by *argv[0]*, *argv[1]*, and so on. The *argv* array must have a null pointer as its last entry. If a C program is being called, its *main* function will see the calling function's *argv[i]* as its *argv[i]*.

The Functions

fexecl is useful when a fixed number of arguments must be passed, and *fexev* when a variable number of arguments must be passed.

Other Information

Files opened for unbuffered i/o in the calling program remain open in the calling program, but are not open in the called program.

SEE ALSO

The EXEC functions also load programs. Since they overlay the calling program, they allow a larger program to be loaded. They never return to the caller.

ERRORS

An *fexec* function returns 0 as its value if it was successful. If it failed, it returns -1 as its value after setting a code in the global integer *errno*. These codes are defined in the Errors section of the Library Overview chapter.

NAME

getenv - Get value of environment variable

SYNOPSIS

char *getenv(name)
char *name;

DESCRIPTION

getenv, returns a pointer to the character string associated with the environment variable *name*, or 0 if the variable isn't in the environment.

The character string is in a dynamically-allocated buffer; this buffer will be released when the next call is made to *getenv*.

Environment variables are set using the *set* command. See the Utility Programs chapter for details.

NAME

mktemp - make a unique file name

SYNOPSIS

```
char *
mktemp (template)
char *template;
```

DESCRIPTION

mktemp replaces the character string pointed at by *template* with the name of a non-existent file, and returns as its value a pointer to the string.

The string pointed at by *template* should look like a file name whose last few characters are *X*s with an optional imbedded period.

mktemp replaces the *X*s with a letter followed by digits. The digits are set to the value that is usually different for each program. The letter will be between 'A' and 'Z', and will be chosen such that the resulting character string isn't the name of an existing file.

DIAGNOSTICS

For a given character string, *mktemp* will try to convert the string into one of 26 file names. If all of these files exist, *mktemp* will replace the first character pointed at by *template* with a null character.

SEE ALSO

tmpfile, tmpnam

EXAMPLES

The following program calls *mktemp* to get a character string that it can use as a file name. If the digits selected by *mktemp* are 1234, then the generated name will be one of the strings *abcA001.234*, *abcB001.234*, *..abcZ001.234*. If all the strings that *mktemp* considers are names of existing files, *mktemp* will replace the first character of the string passed to it, *a* in this case, with 0.

```
#include <stdio.h>
main()
{
    char *fname, *mktemp();
    FILE *fp, fopen();
    fname=mktemp("abcXXX.XXX")==0)
    if (!*fname){
        printf("mktemp failed");
        exit(1);
    } else
        fp=fopen(fname, "w");
    ...
}
```

NAME

perror, errno, sys_errlist, sys_nerr - system error messages

SYNOPSIS

```
int perror (s)
char *s;

#include <errno.h>

extern int errno;

extern char *sys_errlist[];

extern int sys_nerr;
```

DESCRIPTION

When a library function detects an error, it will generally set an error code, which is a positive integer, in the global integer *errno* and return an appropriate, function-dependent value.

sys_errlist is an array of pointers to character strings, each of which is a message corresponding to an *errno* error code. That is, when an error occurs, *errno* can be used as an index into *sys_errlist* to get a message corresponding to the error. The messages don't contain a newline character.

The maximum value that can be placed in *errno*, and the total number of entries in *sys_errlist*, is in the global integer *sys_nerr*.

The *extern* declarations of *errno*, *sys_errlist*, and *sys_nerr* are in *errno.h*.

When an error occurs, *perror* can be called to write a message describing the error on the standard error device. The message consists of the following:

* *s*, the string pointed at by the argument to *perror*,
* a colon and a blank,
* the *sys_errlist* message corresponding to the current value of *errno*,
* a newline character.

perror returns 0 if *errno* contains a valid value; otherwise it returns -1 without printing a message.

SEE ALSO

Error Overview (O)

NAME

scdir -- return the name of the next file matching pattern

SYNOPSIS

```
char * scdir(pat)
char *pat;
```

DESCRIPTION

scdir is a function which permits the user to perform wild card expansion on file name patterns using native Amiga facilities.

When *scdir* is called with a pattern, it returns a pointer to a static area containing the null terminated name of the next file which matches the pattern or zero if no more files match the pattern. Since the area containing the name is statically allocated, the name will be overwritten by subsequent calls to *scdir*.

EXAMPLE

```
main()
{
   char *sav[100];
   register char  *pat;
   register int i;

   /* get all .c files on the current directory */

   pat = "*.c";
   i = 0;

   while ((ptr = scdir(pat)) && i < 100) {
      sav[i] = malloc(strlen(ptr)+1);
      strcpy(sav[i++], ptr);
   }
   /* rest of program */
}
```

NAME

screen manipulation functions:
scr_beep, scr_bs, scr_tab, scr_lf,
scr_cursup, scr_cursrt, scr_cr,
scr_clear, scr_home, scr_curs, scr_eol,
scr_linsert, scr_ldelete,
scr_cinsert, scr_cdelete

SYNOPSIS

scr_beep()

scr_bs()

scr_tab()

scr_lf()

scr_cursup()

scr_cursrt()

scr_cr()

scr_clear()

scr_home()

scr_eol()

scr_linsert()

scr_ldelete()

scr_cinsert()

scr_cdelete()

scr_curs(lin, col)
int lin, col;

DESCRIPTION

These functions can be called by command programs to manipulate screens of text. For example, there are functions to clear the screen, position the cursor, and insert and delete characters and lines.

These functions can be used in conjunction with the normal standard i/o and unbuffered i/o functions to display characters on the console.

A program that calls these functions must access the console using the Aztec console driver; that is, it must have been linked with *shcroot* or *mixcroot*.

scr_beep rings the keyboard bell.

scr_bs moves the cursor back one character space, without modifying the character that was backspaced over.

scr_tab moves the cursor right one tab stop.

scr_lf moves the cursor down one line, scrolling if at the bottom of the screen.

scr_cursup moves the cursor up without changing its column location.

scr_cursrt moves the cursor right one character space, without modifying the character that was spaced over.

scr_cr causes a carriage return.

scr_clear clears the screen and homes the cursor.

scr_home homes the cursor to the upper left hand corner of the screen.

scr_curs moves the cursor to the line and column specified by the *lin* and *col* parameters, respectively.

scr_eol erases the line at which the cursor is located, from the current cursor positon to the end of the line.

scr_linsert inserts a blank line at the cursor location, moving the lines below the cursor down one line.

scr_ldelete deletes the line at the cursor location, moving the lines below the cursor up one line and placing a blank line at the bottom of the screen.

scr_cinsert inserts a space at the cursor location, shifting right one character the characters in the line which are on the right of the cursor.

scr_cdelete deletes the character at the cursor location, shifting left one character the characters in the line which are on the right of the cursor.

NAME

time, ctime, localtime, gmtime, asctime

SYNOPSIS

```
long time(tloc)
long *tloc;

char *ctime(clock)
long *clock;

#include "time.h"

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;
```

DESCRIPTION

time returns the date and time, which it gets from the operating system. The other functions convert the date and time, which are passed as arguments, to another format.

time returns the current date and time packed into a long int. If its argument *tloc* is non-null, the return value is also stored in the field pointed at by the argument. The format of the value returned by *time* is described below.

ctime, *localtime*, and *gmtime* convert a date and time pointed at by their argument, which is in a format such as returned by *time*, to another format:

ctime converts the time to a 26-character ASCII string of the form

Mon Apr 30 10:04:52 1984\n\0

localtime and *gmtime* unpack the date and time into a structure and return a pointer to it. The structure, named *tm*, is described below and defined in the header file *time.h*.

asctime converts a date and time pointed at by its argument, which is in a structure such as returned by *localtime* and *gmtime*, to a 26-character ASCII string in the same form as returned by *ctime*.

The long int returned by *time* and passed to *ctime*, *localtime*, and *gmtime* has the following form (bit 0 is the least significant bit in the field, bit 31 the most significant):

bits	*meaning*
0-4	seconds/2
5-10	minutes
11-15	hours
16-20	day of month
21-24	month (0=Jan,...)
25-31	year since 1980

The long int fields used by the functions described in the FILETIME section also have the above format.

The structure returned by *localtime* and *gmtime*, and passed to *asctime*, has the following format:

```
struct tm {
    short tm_sec;   /* seconds */
    short tm_min;  /* minutes */
    short tm_hour;/* hours */
    short tm_mday;/* day of the month */
    short tm_mon;  /* month */
    short tm_year;  /* year since 1900 */
    short tm_wday;/* day of the week (0 = Sunday */
    short tm_yday;/* day of year */
    short tm_isdst;  /* not used */
    short tm_hsec;   /* hundredths of seconds */
}
```

NAME

tmpfile - create a temporary file

SYNOPSIS

```
#include <stdio.h>
FILE *
tmpfile ()
```

DESCRIPTION

tmpfile creates a temporary file and opens it for standard i/o in update (w+) mode. *tmpfile* returns as its value the file's FILE pointer.

When the temporary file is closed, either because the program explicitly closes it or because the program terminates, the temporary file will automatically be deleted.

SEE ALSO

tmpnam, mktemp

NAME

tmpnam - create a name for a temporary file

SYNOPSIS

char *tmpnam (s)
char *s;

DESCRIPTION

tmpnam creates a character string that can be used as the name of a temporary file and returns as its value a pointer to the string. The generated string is not the name of an existing file.

s optionally points to an area into which the name will be generated. This must contain at least *L_tmpnam* bytes, where *L_tmpnam* is a constant defined in *stdio.h*.

s can also be a NULL pointer. In this case, the name will be generated in an internal array. The contents of this array are destroyed each time *tmpnam* is called with a NULL argument.

The generated name is prefixed with the string that is associated with the symbol *P_tmpnam*; this symbol is defined in *stdio.h*. In the distribution version of *stdio.h*, *P_tmpnam* is a null string; this results in the generated name specifying a file that will be located in the current directory.

SEE ALSO

tmpfile, mktemp

AMIGA FUNCTIONS

Amiga Functions

This chapter supplements the description of the Amiga functions that is in the Amiga manuals, by summarizing the functions' calling sequences when called by Aztec C68k-compiled programs. For each function, this chapter lists the function's parameters and their types, and the type of value returned by the function. Thus, to write programs that call Amiga functions, you should read the Amiga manuals for descriptions of the functions and this chapter for descriptions of the function's calling sequences.

The Amiga manuals describe the Amiga functions' calling sequences, when called by Lattice C-compiled programs. The calling sequences of the Amiga functions is the same for programs compiled with both the Aztec C68k and the Lattice compilers, with one exception: *int*s are 32 bits long for Lattice C-compiled programs, and are by default 16 bits long for Aztec C68k-compiled programs. The Amiga functions expect to be called by Lattice C-compiled programs, which means that an Aztec C68k-compiled program cannot simply pass an *int* variable or constant to an Amiga function that is expecting a 32-bit *int*. Since *long*s are 32 bits long for both Lattice C and Aztec C68k, one way to pass 32 bit integers to Amiga functions is to have your programs pass *long*s instead of *int*s; such a program can then be compiled with either Aztec C68k or Lattice C. This solution is encouraged in this chapter, by listing as a *long* the type of a 32-bit integer that is passed between a program and an Amiga function.

Another way to pass 32-bit integers to Amiga functions is to compile the modules that call Amiga functions with the *+l* option, which causes the module's *int*s to be 32-bits long. We don't recommend this solution, because it slows down a program, requiring *int* arithmetic to be performed on 32-bit instead of 16-bit quantities. One advantage of the use of this option is that it allows a working, Lattice-C compiled program to be recompiled with Aztec C68k without requiring any changes to the program.

long *AbleICR* (mask)
long mask;

void *AbortIO* (ioRequest)
struct IORequest *ioRequest;

void *AddAnimOb* (anOb, anKey, rPort)
struct AnimOb *anOb, **anKey;
struct RastPort *rPort;

void *AddBob* (bob, rPort)
struct Bob *bob;
struct RastPort *rPort;

void *AddDevice* (device)
struct Device *device;

void *AddFont* (textFont)
struct TextFont *textFont;

short *AddFreeList* (free, mem, len)
struct FreeList *free; char *mem; long len;

short *AddGadget* (pntr, gadget, position)
struct Window *pntr; struct Gadget *gadget;
long position;

void *AddHead* (list, node)
struct List *list; struct Node *node;

struct Interrupt * *AddICRVector* (iCRBit, interrupt)
long iCRBit; struct Interrupt *interrupt;

void *AddIntServer* (intNum, interrupt)
long intNum; struct Interrupt *interrupt;

void *AddLibrary* (library)
struct Library *library;

void *AddPort* (port)
struct MsgPort *port;

void *AddResource* (resource)
struct MiscResource *resource;

void *AddTail* (list, node)
struct List *list; struct Node *node;

void *AddTask* (task, initialPC, finalPC)
struct Task *task; long initialPC, finalPC;

void *AddTime* (dest, source)
struct TimeVal *dest, *source;

void *AddVSprite* (vs, rPort)
struct VSprite *vs; struct RastPort *rPort;

long *Alert* (alertNum, parameters)
long alertNum, parameters;

void * *AllocAbs* (bytSiz, location)
long bytSiz; char *location;

cList *AllocCList* (long cLPool)
long cLPool;

struct MemList * *AllocEntry* (memList)
struct MemList *memList;

void * *AllocMem* (byteSize, requirements)
long byteSize, requirements;

long *AllocPotBits* (bits)
long bits;

void *AllocRaster* (width, height)
long width, height;

char * *AllocRemember* (rememberKey, size, flags)
struct Remember *rememberKey; long size, flags;

long *AllocSignal* (signalNum)
long signalNum;

long *AllocTrap* (trapNum)
long trapNum;

struct WBObject * *AllocWBObject* ( )

void * *Allocate* (freeList, byteSize)
struct List *freeList; long byteSize;

void *AlohaWorkbench* (wbPort)
struct MsgPort *wbPort;

void *AndRectRegion* (region, rectangle)
struct Region *region; struct Rectangle *rectangle;

void *Animate* (key, rPort)
struct AnimOb **key; struct RastPort *rPort;

short *AreaDraw* (rp, x, y)
struct RastPort *rp; long x, y;

void *AreaEnd* (rp)
struct RastPort *rp;

short *AreaMove* (rp, x, y)
struct RastPort *rp; long x, y;

```
void AskFont ( rp, textAttr )
        struct RastPort *rp; struct TextAttr *textAttr;

long AskSoftStyle ( rp )
        struct RastPort *rp;

short AutoRequest ( window, body, positive, negative,
        posFlags, negFlags, width, height )
        struct Window *window;
        struct IntuiText *body, *positive, *negative;
        long posFlags, negFlags, width, height;

long AvailFonts ( buffer, bufBytes, types )
        char *buffer; long bufBytes, types;

long AvailMem ( requirements )
        long requirements;

void BeginIO ( ioRequest )
        struct IORequest *ioRequest;

void BeginRefresh ( window )
        struct Window *window;

void BeginUpdate ( l )
        struct Layer *l;

void BehindLayer ( li, l )
        struct LayerInfo *li; struct Layer *l;

long BltBitMap ( srcBM, srcX, srcY, dstBM,
                 dstX, dstY, sizX, sizY,
                 minTerm, mask, char *tempA )
        struct BitMap *srcBM, *dstBM;
        long srcX, srcY, dstX, dstY, sizX, sizY,
           minTerm, mask, char *tempA;

long BltBitMapRastPort ( srcBitMap, srcX, srcY, dstBitMap,
                         dstX, dstY, sizX, sizY, minTerm )
        struct BitMap *srcBitMap, *dstBitMap;
        long srcX, srcY, dstX, dstY, sizX, sizY, minTerm;

void BltClear ( memBlock, byteCount, flags )
        char *memBlock; long byteCount, flags;

void BltPattern ( rp, buf, xl, yl, maxX, maxY, byteCnt )
        struct RastPort *rp; char *buf;
        long xl, yl, maxX, maxY, byteCnt;
```

```
void BltTemplate ( src, srcX, srcMod, dstRastPort,
                   dstX, dstY, sizX, sizY )
        char *src;
        struct RastPort *dstRastPort;
        long srcX, srcMod, dstX, dstY, sizX, sizY;

struct Window * BuildSysRequest ( wind, body, positive, negative,
                                  flags, width, height )
        struct Window *wind;
        struct IntuiText *body, *positive, *negative;
        long flags, width, height;

char * BumpRevision ( newbuf, oldname )
        char *newbuf, *oldname;

void CBump ( c )
        struct UCopList *c;

void CDInputHandler ( events, consoleDevice )
        struct Events *events;
        struct Device *consoleDevice;

void CMove ( c, a, v )
        struct UCopList *c; long *a, v;

void CWait ( c, v, h )
        struct UCopList *c; long v, h;

void Cause ( interrupt )
        struct Interrupt *interrupt;

void ChangeSprite ( vp, s, newdata )
        struct ViewPort *vp; struct SimpleSprite *s;
        struct spriteimage *newdata;

struct IORequest * CheckIO ( ioRequest )
        struct IORequest *ioRequest;

short ClearDMRequest ( window )
        struct Window *window;

void ClearEOL ( rp )
        struct RastPort *rp;

void ClearMenuStrip ( window )
        struct Window *window;

void ClearPointer ( window )
        struct Window *window;

void ClearRegion ( region )
        struct Region *region;
```

```
void ClearScreen ( rp )
        struct RastPort *rp;

void ClipBlit ( src, srcX, srcY, dst,
                dstX, dstY, xSize, ySize, mode )
        struct RastPort *src, *dst;
        long srcX, srcY, dstX, dstY, xSize, ySize, mode;

void Close ( file )
        struct FileHandle *file;

void CloseDevice ( ioRequest )
        struct IORequest *ioRequest;

void CloseFont ( font )
        struct Font *font;

void CloseLibrary ( library )
        struct Library *library;

void CloseScreen ( screen )
        struct Screen *screen;

void CloseWindow ( window )
        struct Window *window;

short CloseWorkBench ( )

short CmpTime ( dest, source )
        struct TimeVal *dest, *source;

long ConcatCList ( sourceCList, destCList )
        cList sourceCList; destCList;

cList CopyCList ( cList )
        cList cList;

void CopySBitMap ( layer )
        struct Layer *layer;

void CreateBehindLayer ( li, bm, x0, y0, x1, y1, flags )
        struct LayerInfo *li; struct BitMap *bm;
        long x0, y0, x1, y1, flags;

struct Lock * CreateDir ( name )
        char *name;

struct MsgPort * CreatePort ( name, pri )
        char *name; long pri;

struct Process * CreateProc ( name, pri, segment, stackSize )
        char *name;
        struct Segment *segment;
        long pri, stackSize;
```

```
struct IOStdReq * CreateStdIO ( mp )
        struct MsgPort *mp;

struct Task * CreateTask ( name, pri, start_pc, stksiz )
        char *name; long pri, start_pc, stksiz;

void CreateUpfrontLayer ( li, bm, x0, y0, x1, y1, flags )
        struct LayerInfo *li; struct BitMap *bm;
        long x0, y0, x1, y1, flags;

struct Lock * CurrentDir ( lock )
        struct Lock *lock;

void CurrentTime ( seconds, micros )
        unsigned long *seconds, *micros;

long * DateStamp ( v )
        long *v;

void Deallocate ( freeList, memoryBlock, byteSize )
        struct List *freeList;
        char *memoryBlock; long byteSize;

void Debug ( )

void Delay ( timeout )
        long timeout;

short DeleteFile ( name )
        char *name;

void DeleteLayer ( li, l )
        struct LayerInfo *li; struct Layer *l;

void DeletePort ( port )
        struct MsgPort *port;

void DeleteStdIO ( iop )
        struct IOStdReq *iop;

void DeleteTask ( tp )
        struct Task *tp;

struct Process * DeviceProc ( name )
        char *name;

void Disable ( )

void DisownBlitter ( )

short DisplayAlert ( alertNumber, string, height )
        long alertNumber; char *string; long height;

void DisplayBeep ( screen )
        struct Screen *screen;
```

```
void DisposeRegion ( region )
        struct Region *region;

void DoCollision ( rPort )
        struct RastPort *rPort;

long DoIO ( ioRequest )
        struct IORequest *ioRequest;

short DoubleClick ( startSeconds, startMicros,
                    currentSeconds, currentMicros )
        unsigned long startSeconds, startMicros,
                    currentSeconds, currentMicros;

void Draw ( rp, x, y )
        struct RastPort *rp; long x, y;

void DrawBorder ( rp, b, leftOffset, topOffset )
        struct RastPort *rp;
        struct Border *b;
        long leftOffset, topOffset;

void DrawGList ( rPort, vPort )
        struct RastPort *rPort; struct ViewPort *vPort;

void DrawImage ( rp, image, leftOffset, topOffset )
        struct RastPort *rp;
        struct Image *image;
        long leftOffset, topOffset;

struct Lock * DupLock ( lock )
        struct Lock *lock;

void Enable ( )

void EndRefresh ( window, complete )
        struct Window *window; long complete;

void EndRequest ( req, wind )
        struct Requester *req; struct Window *wind;

void EndUpdate ( l, flag )
        struct Layer *l; long flag;

void Enqueue ( list, node )
        struct List *list, struct Node *node;

short ExNext ( lock, fileInfoBlock )
        struct Lock *lock;
        struct FileInfoBlock *fileInfoBlock;

short Examine ( lock, fileInfoBlock )
        struct Lock *lock;
        struct FileInfoBlock *fileInfoBlock;
```

```
short Execute ( commandString, input, output )
        char *commandString;
        struct FileHandle *input, *output;

void Exit ( returnCode )
        long returnCode;

struct Node * FindName ( list, name )
        struct List *list; char *name;

struct MsgPort * FindPort ( name )
        char *name;

struct Task * FindTask ( name )
        char *name;

char * FindToolType ( toolTypeArray, typeName )
        char **toolTypeArray, *typeName;

void Flood ( rp, mode, x, y )
        struct RastPort *rp; long mode, x, y;

void FlushCList ( cList )
        cList cList;

void Forbid ( )

void FreeCList ( cList )
        cList cList;

void FreeColorMap ( colorMap )
        struct ColorMap *colorMap;

void FreeCopList ( copList )
        struct CopList *copList;

void FreeCprList ( cprList )
        struct CprList *cprList;

void FreeDiskObject ( diskobj )
        struct DiskObject *diskobj;

void FreeEntry ( memList )
        struct MemList *memList;

void FreeFreeList ( free )
        struct FreeList *free;

void FreeGBuffers ( anOb, RPort, db )
        struct AnimOb *anOb;
        struct RastPort *Rport;
        long db;

void FreeMem ( memoryBlock, sizeBytes )
        char *memoryBlock; long sizeBytes;
```

void *FreePotBits* (allocated)
long allocated;

void *FreeRaster* (p, width, height)
char *p; long width, height;

void *FreeRemember* (key, reallyForget)
struct Remember *key; long reallyForget;

void *FreeSignal* (signalNum)
long signalNum;

void *FreeSprite* (pick)
long pick;

void *FreeSysRequest* (wind)
struct Window *wind;

void *FreeTrap* (trapNum)
long trapNum;

void *FreeVPortCopLists* (viewPort)
struct ViewPort *viewPort;

void *FreeWBObject* (obj)
struct WBObject *obj;

long *GetCC* ( )

long *GetCLBuf* (cList, buffer, maxlength)
cList cList; char *buffer; long maxlength;

short *GetCLChar* (cList)
cList cList;

short *GetCLWord* (cList)
cList cList;

struct ColorMap * *GetColorMap* (entries)
long entries;

struct Preferences * *GetDefPrefs* (prefBuffer, size)
char *prefBuffer; long size;

struct DiskObject * *GetDiskObject* (name)
char *name;

void *GetGBuffers* (anOb, rPort, db)
struct AnimOb *anOb;
struct RastPort *rPort;
long db;

```
short GetIcon ( name, icon, free )
        char *name;
        struct DiskObject *icon;
        struct FreeList *free;

struct Message * GetMsg ( port )
        struct MsgPort *port;

struct Preferences * GetPrefs ( buffer, size )
        struct Preferences *buffer; long size;

short GetRGB4 ( colorMap, entry )
        struct ColorMap *colorMap; long entry;

short GetSprite ( sprite, pick )
        struct SimpleSprite *sprite; long pick;

struct WBObject * GetWBObject ( name )
        char *name;

long IncrCLMark ( cList )
        cList cList;

short Info ( lock, info_Data )
        struct Lock *lock; struct Info_Data *info_Data;

void InitArea ( areaInfo, buffer, maxVectors )
        struct AreaInfo *areaInfo;
        char *buffer; long maxVectors;

void InitBitMap ( bm, depth, width, height )
        struct BitMap *bm; long depth, width, height;

long InitCLPool ( cLPool, size )
        cLPool *cLPool; long size;

void InitCode ( startClass, version )
        long startClass, version;

void InitGMasks ( anOb )
        struct AnimOb *anOb;

void InitGels ( head, tail, gInfo )
        struct VSprite *head;
        struct VSprite *tail;
        struct GelsInfo *gInfo;

void InitMasks ( vs )
        struct VSprite *vs;

void InitRastPort ( rp )
        struct RastPort *rp;

void InitRequester ( req )
        struct Requester *req;
```

```
void InitResident ( resident, segList )
        long resident, segList;

void InitStruct ( initTable, memory, size )
        short *initTable; char *memory; long size;

void InitTmpRas ( tmpRas, buffer, size )
        struct TmpRas *tmpRas; char *buffer; long size;

void InitVPort ( vp )
        struct ViewPort *vp;

void InitView ( view )
        struct View *view;

struct FileHandle * Input ( )

void Insert ( list, node, listNode )
        struct List *list; struct Node *node, *listNode;

long IntuiTextLength ( iText )
        struct IntuiText *iText;

struct InputEvent * Intuition ( inputEvent )
        struct InputEvent *inputEvent;

long IoErr ( )

short IsInteractive ( file )
        struct FileHandle *file;

struct MenuItem * ItemAddress ( menuStrip, menuNumber )
        struct Menu *menuStrip; long menuNumber;

void LoadRGB4 ( vp, colorMap, count )
        struct ViewPort *vp;
        struct ColorMap *colorMap;
        long count;

struct Segment * LoadSeg ( name )
        char *name;

void LoadView ( view )
        struct View *view;

struct Lock * Lock ( name, accessMode )
        char *name; long accessMode;

void LockLayer ( li, l )
        struct LayerInfo *li; struct Layer *l;

void LockLayerInfo ( li )
        struct LayerInfo *li;

void LockLayerRom ( layer )
        struct Layer *layer;
```

```
void LockLayers ( li )
        struct LayerInfo *li;

struct Library * MakeLibrary ( vectors, structure,
                               libInit, dataSize, segList )
        void (*vectors[])(), (*libInit)();
        short *structure;
        long dataSize;
        struct MemList *segList;

void MakeScreen ( screen )
        struct Screen *screen;

void MakeVPort ( view, viewPort )
        struct View *view; struct ViewPort *viewPort;

long MarkCList ( cList, offset )
        cList cList; long offset;

long MatchToolValue ( typeString, value )
        char *typeString, *value;

void ModifyIDCMP ( wind, IDCMPFlags )
        struct Window *wind; long IDCMPFlags;

void ModifyProp ( gad, wind, req, flags,
                  horizPot, vertPot, horizBody, verBody )
        struct Gadget *gad;
        struct Window *wind;
        struct Requester *req;
        long flags, horizPot, vertPot, horizBody, verBody;

void Move ( rp, x, y )
        struct RastPort *rp; long x, y;

void MoveLayer ( li, l, dx, dy )
        struct LayerInfo *li; struct Layer *l; long dx, dy;

void MoveScreen ( screen, deltaX, deltaY )
        struct Screen *screen; long deltaX, deltaY;

void MoveSprite ( vp, sprite, x, y )
        struct ViewPort *vp;
        struct SimpleSprite *sprite;
        long x, y;

void MoveWindow ( wind, deltaX, deltaY )
        struct Window *wind; long deltaX, deltaY;

void MrgCop ( view )
        struct View *view;

void NewList ( list )
        struct List *list;
```

```
struct Region * NewRegion ( )

void NotRegion ( rgn )
        struct Region *rgn;

void OffGadget ( gad, wind, req )
        struct Gadget *gad;
        struct Window *wind;
        struct Requester *req;

void OffMenu ( wind, menuNumber )
        struct Window *wind; long menuNumber;

void OnGadget ( gad, wind, req )
        struct Gadget *gad;
        struct Window *wind;
        struct Requester *req;

void OnMenu ( wind, menuNumber )
        struct Window *wind; long menuNumber;

struct FileHandle * Open ( name, accessMode )
        char *name; long accessMode;

long OpenDevice ( name, unitNumber, ioRequest, flags )
        char *name;
        struct IORequest *ioRequest;
        long unitNumber, flags;

struct Font * OpenDiskFont ( textAttr )
        struct TextAttr *textAttr;

struct Font * OpenFont ( textAttr )
        struct TextAttr *textAttr;

void OpenIntuition ( )

struct Library * OpenLibrary ( libName, version )
        char *libName; long version;

struct MiscResource * OpenResource ( resName )
        char *resName;

struct Screen * OpenScreen ( newScreen )
        struct NewScreen *newScreen;

struct Window * OpenWindow ( newWindow )
        struct NewWindow *newWindow;

short OpenWorkBench ( )

void OrRectRegion ( region, rectangle )
        struct Region *region;
        struct Rectangle *rectangle;
```

```
struct FileHandle * Output ( )

void OwnBlitter ( )

struct Lock * ParentDir ( lock )
        struct Lock *lock;

short PeekCLMark ( cList )
        cList cList;

void Permit ( )

void PolyDraw ( rp, count, array )
        struct RastPort *rp;
        long count, array[][2];

void PrintIText ( rp, iText, leftEdge, topEdge )
        struct RastPort *rp;
        struct IntuiText *iText;
        long leftEdge, topEdge;

long PutCLBuf ( cList, buffer, length )
        cList cList; char *buffer; long length;

long PutCLChar ( cList, byte )
        cList cList; long byte;

long PutCLWord ( cList, word )
        cList cList; long word;

short PutDiskObject ( name, diskobj )
        char *name; struct DiskObject *diskobj;

short PutIcon ( name, icon )
        char *name; struct DiskObject *icon;

void PutMsg ( port, message )
        struct MsgPort *port; struct Message *message;

short PutWBObject ( name, object )
        char *name; struct WBObject *object;

void QBSBlit ( bsp )
        struct Blit *bsp;

void QBlit ( bp )
        struct Blit *bp;

short RawKeyConvert ( events, buffer, length, keyMap )
        struct InputEvent *events; char *buffer;
        long length; struct KeyMap *keyMap;

long Read ( file, buffer, length )
        struct FileHandle *file; char *buffer; long length;
```

```
short ReadPixel ( rp, x, y;
        struct RastPort *rp; long x, y;

void RectFill ( rp, xMin, yMin, xMax, yMax )
        struct RastPort *rp;
        long xMin, yMin, xMax, yMax;

void RefreshGadgets ( gad, wind, req )
        struct Gadgets *gad;
        struct Window *wind;
        struct Requester *req;

void RemDevice ( device )
        struct Device *device;

void RemFont ( textFont )
        struct TextFont *textFont;

struct Node * RemHead ( list )
        struct List *list;

void RemIBob ( bob, rPort, vPort )
        struct Bob *bob;
        struct RastPort *rPort;
        struct ViewPort *vPort;

void RemICRVector ( iCRBit, interrupt )
        long iCRBit; struct Interrupt *interrupt;

void RemIntServer ( intNum, interrupt )
        long intNum; struct Interrupt *interrupt;

long RemLibrary ( library )
        struct Library *library;

void RemPort ( port )
        struct MsgPort *port;

void RemResource ( resource )
        struct MiscResource *resource;

struct Node * RemTail ( list )
        struct List *list;

void RemTask ( task )
        struct Task *task;

void RemVSprite ( vs )
        struct VSprite *vs;

void RemakeDisplay ( )

void Remove ( list, node )
        struct List *list; struct Node *node;
```

unsigned short *RemoveGadget* (wind, gad)
struct Window *wind; struct Gadget *gad;

short *Rename* (oldName, newName)
char *oldName, *newName;

void *ReplyMsg* (message)
struct Message *message;

void *ReportMouse* (wind, boolean)
struct Window *wind; long boolean;

short *Request* (req, wind)
struct Requester *req; struct Window *wind;

void *RethinkDisplay* ( )

void *ScreenToBack* (screen)
struct Screen *screen;

void *ScreenToFront* (screen)
struct Screen *screen;

void *ScrollLayer* (li, l, dx, dy)
struct LayerInfo *li;
struct Layer *l; long dx, dy;

void *ScrollRaster* (rp, dx, dy, xMin, yMin, xMax, yMax)
struct RastPort *rp;
long dx, dy, xMin, yMin, xMax, yMax;

void *ScrollVPort* (vp)
struct ViewPort *vp;

long *Seek* (file, position, mode)
struct FileHandle *file; long position, mode;

void *SendIO* (ioRequest)
struct IORequest *ioRequest;

void *SetAPen* (rp, pen)
struct RastPort *rp; long pen;

void *SetBPen* (rp, pen)
struct RastPort *rp; long pen;

void *SetCollision* (num, routine, gInfo)
long num, (*routine)(); struct GelsInfo *gInfo;

short *SetComment* (name, comment)
char *name, *comment;

short *SetDMRequest* (wind, DMRequester)
struct Window *wind;
struct Requester *DMRequester;

void *SetDrMd* (rp, mode)
struct RastPort *rp; long mode;

long *SetExcept* (newSignals, signalMask)
long newSignals, signalMask;

long *SetFont* (rp, font)
struct RastPort *rp; struct TextFont *font;

long *SetFunction* (library, funcOffset, funcEntry)
struct Library *library;
long funcOffset;
void (*funcEntry)();

long *SetICR* (mask)
long mask;

struct Interrupt * *SetIntVector* (intNumber, interrupt)
long intNumber; struct Interrupt *interrupt;

void *SetMenuStrip* (wind, menu)
struct Window *wind; struct Menu *menu;

void *SetPointer* (wind, sp, height, width, xOffset, yOffset)
struct Window *wind; struct Sprite *sp;
long height, width, xOffset, yOffset;

void *SetPrefs* (p, size, realThing)
struct Preferences *p; long size, realThing;

short *SetProtection* (name, mask)
char *name; long mask;

void *SetRGB4* (vp, n, r, g, b)
struct ViewPort *vp; long n, r, g, b;

void *SetRast* (rp, pen)
struct RastPort *rp; long pen;

long *SetSR* (newSR, mask)
long newSR, mask;

long *SetSignal* (newSignals, signalMask)
long newSignals, signalMask;

long *SetSoftStyle* (rp, style, enable)
struct RastPort *rp; long style, enable;

short *SetTaskPri* (task, priority)
struct Task *task; long priority;

void *SetWindowTitles* (wind, windowTitle, screenTitle)
struct Window *wind;
char *windowTitle, *screenTitle;

void *ShowTitle* (screen, showIt)
struct Screen *screen; long showIt;

void *Signal* (task, signals)
struct Task *task; long signals;

long *SizeCList* (cList)
cList cList;

void *SizeLayer* (li, l, dx, dy)
struct LayerInfo *li;
struct Layer *l;
long dx, dy;

void *SizeWindow* (wind, deltaX, deltaY)
struct Window *wind; long deltaX, deltaY;

void *SortGList* (rPort)
struct RastPort *rPort;

cList *SplitCList* (cList)
cList cList;

cList *SubCList* (cList, index, length)
cList cList; long index, length;

void *SubTime* (dest, source)
struct TimeVal *dest, *source;

void *SumLibrary* (library)
struct Library *library;

long *SuperState* ( )

void *SwapBitsRastPortClipRect* (li)
struct LayerInfo *li;

void *SyncSBitMap* (layer)
struct Layer *layer;

long *Text* (rp, string, count)
struct RastPort *rp;
char *string; long count;

long *TextLength* (rp, string, count)
struct RastPort *rp; char *string; long count;

long *Translate* (instring, inlen, outbuf, outlen)
char *instring, *outbuf; long inlen, outlen;

long *UnGetCLChar* (cList, byte)
cList cList; long byte;

long *UnGetCLWord* (cList, word)
cList cList; long word;

```
void UnLoadSeg ( segment )
        struct Segment *segment;

void UnLock ( lock )
        struct Lock *lock;

short UnPutCLChar ( cList )
        cList cList;

short UnPutCLWord ( cList )
        cList cList;

void UnlockLayer ( l )
        struct Layer *l;

void UnlockLayerInfo ( li )
        struct LayerInfo *li;

void UnlockLayerRom ( layer )
        struct Layer *layer;

void UnlockLayers ( li )
        struct LayerInfo *li;

void UpfrontLayer ( li, l )
        struct LayerInfo *li; struct Layer *l;

void UserState ( sysStack )
        long sysStack;

short VBeamPos ( )

struct View * ViewAddress ( )

struct View * ViewPortAddress ( wind )
        struct Window *wind;

short WBenchToBack ( )

short WBenchToFront ( )

long Wait ( signalSet )
        long signalSet;

void WaitBOVP ( viewPort )
        struct ViewPort *viewPort;

void WaitBlit ( )

short WaitForChar ( file, timeout )
        struct FileHandle *file; long timeout;

long WaitIO ( ioRequest )
        struct IORequest *ioRequest;

struct Message * WaitPort ( port )
        struct MsgPort *port;
```

```
void WaitTOF ( )

struct Layer * WhichLayer ( li, x, y )
        struct LayerInfo *li; long x, y;

short WindowLimits ( wind, minWidth, minHeight,
                     maxWidth, maxHeight )
        struct Window *wind;
        long minWidth, minHeight, maxWidth, maxHeight;

void WindowToBack ( wind )
        struct Window *wind;

void WindowToFront ( wind )
        struct Window *wind;

long Write ( file, buffer, length )
        struct FileHandle *file; char *buffer; long length;

void WritePixel ( rp, x, y )
        struct RastPort *rp; long x, y;

void WritePotgo ( word, mask )
        long word, mask;

void XorRectRegion ( region, rectangle )
        struct Region *region;
        struct Rectangle *rectangle;
```

TECHNICAL INFORMATION

Chapter Contents

Technical Information ..... tech
Program Organization 4
Code Segmentation 5
Libraries 8
Interfacing to Assembly Language 9
Interrupt Handlers 12

Technical Information

This chapter discusses technical topics, and topics that couldn't be conveniently discussed elsewhere.

It's divided into the following sections:

1. *Program Organization.* Discusses the factors that affect the memory organization of a program.
2. *Code Segmentation.* Describes the partitioning of a program's executable code into several segments.
3. *Libraries.* Discusses the object module libraries that are provided with Aztec C68k.
4. *Mixing Assembler and C Routines.* Describes how to interface assembly language routines with C routines.
5. *Interrupt Handlers*

1. Program Organization

A program is organized into the following segments:

* one or more *code segments*, containing the program's executable code;
* an *initialized data segment*;
* an *uninitialized data segment*;
* a *stack segment*;
* one or more segments that are dynamically-allocated for use as I/O buffers, etc.

The following paragraphs discuss these segments.

1.1 Placement of Segments in Memory

The placement in memory of a program segment is independent of the placement of its other segments, with one exception: if any of the program's modules uses the 'small data' memory model, then the program's initialized and uninitialized data segments occupy a single, contiguous block of memory, with the uninitialized data following the initialized.

The linker +C and +F options can be used to force a program's code, initialized data, and/or uninitialized data segments to be loaded into chip or fast memory. For details, see the Linker chapter.

Memory segments that are dynamically allocated by calling the UNIX-compatible functions (eg, *malloc*) are allocated from whatever memory is available. The Amiga *AllocMem* function can be called to allocate memory from a specific area.

1.2 Segment size

There is no limit on the size of a program's code segments, regardless of the memory model used by the program's modules.

If all a program's modules use the 'large data' memory model, then there is no limit to the sizes of its initialized and uninitialized data segments. If any of its modules uses the 'small data' memory model, then register A4 points into the middle of the program's data area, and data that 'small data' modules attempt to access must lie within 32k bytes on either side of the location pointed at by A4.

When a program is loaded within the CLI environment, the size of its stack area is set to the value specified in the most recently-entered *stack* command, or to the CLI default value, if a *stack* command hasn't been entered. When a program is loaded by the Workbench, the program's *.info* file defines the size of its stack area.

For a discussion of the size of dynamically-allocated buffers, see the Dynamically-Allocated Memory section in the chapter entitled Library Overview: Amiga Information.

2. Segmented Code

With Aztec C68k, you can create and execute programs that are larger than available memory, by dividing a program's executable code into several segments. A segment is brought into memory when needed; when it's no longer needed, the memory it occupied can be released and reused.

This section describes the creation and use of segmented programs. It's divided into the following paragraphs:

* *Programming Information.* Describes how you write programs having segmented code.
* *Operator Information.* Describes how you link programs having segmented code.

2.1 Programmer Information

There are two areas of concern for programs whose code is segmented: how segments are loaded and unloaded, and how programs access global and static data.

2.1.1 Loading and unloading code segments

A function can call another function without regard for the segments that contain the two functions: if they are in the same segment, or if they are in different segments and the other segment is already in memory, the call will be immediately made; if they are in different segments and the other segment isn't in memory, the other segment will automatically be loaded and then the call will be performed.

One code segment of a program is called the "root segment"; its segment number is 0 (see below for details), and it contains the program's entry point. When a program is started, its root code segment and its data segment are loaded into memory, and control is transfered to the program's entry point. The program's other code segments will be automatically loaded into memory when they're needed.

Once loaded, a code segment will remain in memory. The program can explicitly unload it, if desired, by calling the *UnloadSeg* function. When the program terminates, all its loaded segments will be automatically released.

2.1.2 Global and static data

There is only one global and static data segment for a program, regardless of the number of code segments it has. This segment is loaded automatically along with the program's root segment, and remains in memory during the entire execution of the program, independent of the state of the program's code segments.

The name of each global variable in a program is unique: if several segments declare a global variable having the same name, they will both access the same variable.

2.2 Operator Information

All of a program's code segments are created during a single activation of the linker.

The code for a command program can be divided into a maximum of 256 segments, each of which has an identifying number between 0 and 255. All command programs must have a code segment 0, which is the first segment loaded for the program.

The linker command which creates a command program whose code is segmented looks like the command which links an unsegmented program, except that the list of files are interspersed with '+O' options. This option causes the object modules which follow it to be placed in a selected code segment.

A segment number can optionally be appended to a '+O' option, to explicitly select the segment into which the following modules will be placed. If a segment number isn't specified for a '+O' option, the modules will be placed in the next available segment.

An example

For example, the following command creates the command program *prog*, which has three code segments. Segment 0 contains the code for the modules *menu.o, subs.o,* and any needed modules from *c.lib.* Segment 1 contains the code for the modules *mod1.o* and *mod2.o.* Segment 2 contains the code for *mod3.o* and *mod4.o,* and any *c.lib* modules referenced by segments 1 and 2 which aren't in segment 0:

```
ln -f prog.lnk
```

where *prog.lnk* contains:

```
menu.o subs.o -lc
+o
mod1.o mod2.o
+O
mod3.o
mod4.o
-lc
```

All the files for this example could have been specified on the command line which activated the linker. We didn't do this for two reasons: first, the entire command wouldn't have fit on one line of this page. Second, you'll also want to use -F files to link programs that have segmented code, since such programs usually have many modules, making it impractical to specify all the file and segmentation information on one line.

Including modules from Libraries

This example illustrates a point about library searches during the linking of command programs having segmented code. As the linker includes object modules in segments, it builds a list of global symbols that are called or referenced but haven't been found yet. When a library is searched during the linking of a segment, and a module is found that contains a needed global symbol, the module is included in the segment, regardless of the segment which referenced it.

Thus, in the above example, when *c.lib* is searched during the linking of segment 0, the only modules in it that are included in segment 0 are those referenced by segment 0. When *c.lib* is searched during the linking of segment 2, modules from it are included that contain global symbols referenced by both segment 1 and 2 but that aren't in segment 0.

Segment 0 must contain the startup code for a program. This code is in *c.lib;* hence *c.lib* must always be first searched while segment 0 is selected. It would not be correct, for example, to modify the above example so that *c.lib* was searched only during the linking of segment 2, since this would force the startup code to be placed in segment 2.

Reselecting segments

The linker allows segments to be selected once, and later be reselected. In this case, the modules specified following the reselection are appended to the code that's already in the segment. A segment can be reselected any number of times.

For example, the above example can be modified so that all *c.lib* modules referenced by all segments are included in segment 0, by modifying *prog.lnk* as follows:

```
menu.o subs.o
+O
mod1.o mod2.o
+O
mod3.o mod4.o
+O0
-lc
```

The '+O0' reselects segment 0, so that the modules pulled from *c.lib* are included in segment 0.

3. Object Module Libraries

Several libraries of object modules are provided with Aztec C68k, the modules of which all use the 'small code', 'small data' memory model.

Two versions of each library are supplied, one of which uses 16 bit *int*s, the other 32 bit *int*s. The libraries using 16 bit *int*s are:

c.lib	non-floating point functions
m.lib	floating point functions
s.lib	screen functions

The libraries using 32 bit *int*s are:

c32.lib	non-floating point functions
m32.lib	floating point functions
s32.lib	screen functions

4. Assembly-Language Functions

This section discusses assembly-language functions that can be called by, and themselves call, C-language functions. It first discusses the conventions that such functions must follow, and then discusses the in-line placement of assembly-language statements within C-language functions.

4.1 Conventions for C-callable, assembly-language functions

A C-callable, assembly-language function must obey the conventions that are described in the following paragraphs.

4.1.1 Global variables

A C module's global variables are in either the uninitialized data segment or in the initialized data segment.

An assembly language module can create an uninitialized variable that can be accessed by a C function, using the *global* directive. For example, the following code creates the global variable *_var*, which can be accessed as an array by a C function, and reserves 8 bytes of storage for it.

```
global _var,8
```

A C function that wants to access *_var* could have the following declaration:

```
extern short var[];
```

An assembly language module can create an initialized variable that can be accessed by a C function using the *public* directive and the *dc* directive. For example, the following code creates the *public* variable *_ptr* that initially contains a pointer to the symbol *str*, and that can be accessed as a *char* pointer by a C function:

```
        dseg
        public  ptr
_ptr    dc.l    str
```

To access *_ptr*, a C function could use the following declaration:

```
extern char *ptr;
```

An assembly language module can access global initialized or uninitialized variables that are created in C modules by defining the variables using a *public* directive that is in the *dseg* segment. For example, suppose a C module creates a global, uninitialized *short* named *count* and a global, initialized *short* named *total* using the statement:

```
short count, total=1;
```

An assembly language module can access these variables by using the following directives:

```
        dseg
        public  _count, _total
```

The above discussion assumes that the C modules and the assembly language modules follow the standard rule in the C language regarding external variables. This rule requires a global variable to be defined without the *extern* keyword in exactly one module, and with that keyword in all other modules. Aztec C also supports a relaxed version of this rule. For information on this, see the discussion on External variables in the Programmer Information section of the Compiler chapter, and the description of the *-M* option in the Linker chapter.

4.1.2 Names of external functions and variables

The C compiler translates the name of a function or variable to assembly language by truncating the name to 31 characters and then preending an underscore character. Thus, assembly language modules that are to be accessible from C-language modules or that are to access C modules must obey this convention.

For example, the following C language module calls the function *bmp*, which simply adds 10 to the global *short count*. A C-language module refers to this function as *bmp*, and an assembly-language module refers to it as *_bmp*.

```
int count;
main()
{
        bmp();
}
```

An assembly language version of *_bmp* could be:

```
        dseg
        public _count
        cseg
        public  _bmp
_bmp:
        add.w   #10,_count
        rts
        end
```

4.1.3 Function calls and returns

The assembly language code generated by the compiler for a C language call to another function pushes the arguments onto the stack, in the reverse order in which they were specified in the call's argument list, and then calls the function.

An assembly language function returns to a C function caller by issuing a *rts* instruction, and leaving the caller's arguments on the stack. The caller then removes the arguments from the stack.

A function returns an integer or pointer in register D0. Floating point values are returned in internal memory locations, and are not discussed here.

For example, consider the following assembly language function, *_sub*, that takes two *short* arguments that are passed to it on the stack, subtracts them, and returns the difference as the function value. A C-language function will refer to this function using the name *sub*.

```
        cseg
        public  _sub
_sub:
        mov     4(sp),d0        ;get first argument
        sub     6(sp),d0        ;subtract second from first
        rts
```

The following C function calls *sub* to subtract *b* from *a*, and stores the difference in *c*:

```
main()
{
        short a,b,c;
        ...
        c = sub(a,b);
}
```

4.1.4 Register usage

An assembly language function that is called by a C function must preserve all registers it uses, except for D0-D3, A0, A1, and A6.

4.2 Embedded Assembler Source

Assembly language statements can be embedded in a "C" program between an *#asm* and an *#endasm* statement. The pound sign (#) must stand in column one of the line, and the letters must be lower case.

Embedded assembler code must preserve the contents of all registers it uses, except for D0-D3, A0, A1, and A6.

It should make no assumptions about the contents of the registers, since the code that the compiler currently generates for C statements may change in the future.

To be safe, a *#asm* statement should be preceded by a semicolon. This avoids problems in which the compiler mistakenly puts a label that is the target of a jump statement after, rather than before, in-line assembly code.

In general, it is safest to contain assembly code in a separate, assembly-language, module rather than embedding it in C source.

5. Interrupt Handlers

An interrupt handler can be written in C, with one caveat: if it uses the small code or small data memory model, it must insure that register A4 points 32766 bytes after the beginning of the program's initialized data segment.

One way to do this is as follows:

* In the Interrupt structure that is passed to the *SetIntVector* function to define the interrupt handler, set the structure's *is_Data* field to the value *_Dorg*+32766, where *_Dorg* is defined in your program to be a function. (The linker creates this symbol for every program).
* In the interrupt handler, before anything else is done, have in-line assembly-language code move register D1 (which contains the Interrupt structure's is_Data field) to register A4.

For example, the part of the code that sets up the interrupt could look like this:

```
int _Dorg();
struct Interrupt xx;

setupint()
{
        ...
        xx.is_Code = intfunc;
        xx.is_Data = &_Dorg;
        xx.is += 32766;
        ...
```

And the beginning of the interrupt handler *intfunc* could look like this:

```
intfunc()
{
#asm
        move.l d1,a4
#endasm
        ...
```

DEBUGGING UTILITIES

Chapter Contents

Debugging Utilities ..... debug
db (program debugger) 4
1. Overview 5
1.1 Basic Commands 5
1.2 Names 5
1.2.1 Code and Data Symbols 6
1.2.2 Operator Usage of Names 6
1.3 Loading programs and symbols 6
1.4 Breakpoints 7
1.5 Memory-change breakpoints 8
1.6 Trace mode 8
1.7 Backtracing 8
1.8 Macros 9
1.9 Other features 9
2. Using DB 10
2.1 Starting DB 10
2.2 Termination 10
2.3 Support for Scatter-Loaded programs 10
2.4 Support for Segmented Programs (Overlays) 11
2.5 Input/Output 11
2.6 Commands 11
2.6.1 Definitions 11
2.7 Command descriptions 15
2.7.1 The AMIGA (a) commands 16
2.7.2 The BREAKPOINT (b) commands 20
2.7.3 The CLEAR (c) commands 24
2.7.4 The DISPLAY (d) commands 25
2.7.5 The GO (g) commands 27
2.7.6 The LOAD (l) commands 28
2.7.7 The MODIFY MEMORY (m) commands 28
2.7.8 The Radix (n) command 30
2.7.9 The PRINT (p) command 30
2.7.10 The QUIT (q) command 36
2.7.11 The REGISTER (r) command 37
2.7.12 The SINGLE STEP (s/t) commands 37
2.7.13 The UNASSEMBLE (u) commands 38
2.7.14 The VARIABLE (v) commands 38
2.7.15 The MACRO (x) command 39
2.7.16 The 'Display Expression' command 39
2.7.17 The 'Redirect command input/output' command . 40
2.7.18 The HELP (?) command 40
3. Command Summary 41

Debugging Utilities

This chapter describes the debugger utility *db* that is provided with some versions of Aztec C68.

NAME

db - symbolic debugger

SYNOPSIS

db

DESCRIPTION

db is used to debug programs which have been created using the Aztec C compiler, assembler, and linker.

db has all the standard features of an assembly language debugger. It also has features not found in all debuggers, such as the ability to reference memory locations by name as well as by address, the ability to define sequences of commands to be macros, which can then be activated by entering a single letter, and a flexible mechanism for handling breakpoints.

In addition, *db* has features specifically tailored to its use with Aztec C, such as the ability to list the name and parameters of the currently executing function, and the function that called it, and so on, back to the initial function. Another special feature is the ability to display, on entry and exit from each function, the function's parameters and return value. Finally, *db* supports both scatter-loaded and segmented programs.

Requirements

An Amiga with at least 512K of memory is recommended for use with *db*. The debugger itself uses about 96K.

Preview

The remainder of this description of *db* is in three sections: *overview*, which describes *db* features in more detail and introduces the commands; *usage*, which describes in full detail how to use *db*; and a *command summary*.

1. Overview

db commands consist of one or two characters, the first of which identifies the command category. If there's only one command in the category, then the command has just this one letter; otherwise, the command has a second letter which identifies the specific operation to be performed.

1.1 Basic commands

db has two types of commands for examining memory: display and print, whose first characters are *d* and *p*, respectively. The 'display' commands *db* and *dw* simply display hexadecimal bytes and words.

The 'print' command, *p*, is more powerful, being able to convert a sequence of one or more possibly different types of data items to ASCII. For example, you can tell it that beginning at the location *var* are a sequence of the following items: an *int*, a *float*, and a pointer to a *char* string. The *p* command will convert the two binary items to ASCII and print them, and display the referenced character string.

The 'register' command, *r*, displays and modifies the 68000 registers.

The 'memory modify' commands, *m*, modify memory.

The *u* commands 'unassemble' code; that is, display it symbolically, in a form similar to its appearance in an assembly language source file.

The *s*, *t* and *g* commands cause the user's program to be executed. *s* and *t* commands "single step" the user's program; that is, execute a specified number of instructions in the user's program and then return control to *db*. The *t* command differs from the *s* command by single stepping over *jsr* and *bsr* instructions. *g* commands transfer control of the processor unconditionally to the user's program. In this case, *db* regains control when the user's program terminates, when an error occurs (such as division by zero), or when a "breakpoint" is taken. Breakpoints are discussed below.

? is the *help* command: it causes *db* to display a summary of all *db* commands. For some command categories, you can get information about the commands in a category by typing the first letter of the category's commands followed by a *?*. For example, typing *m?* gets you information about the memory modification commands (all of whose first letter is *m*).

1.2 Names

db allows memory locations to be referenced by name as well as by location. It learns a program's global names by reading the symbols from the program file and placing them in a memory-resident symbol table. The linker generates a symbol table hunk for a program in response to the "-w" option.

db only allows **global** symbols to be accessed by name; automatic variables and static variables can't be accessed by name.

The operator can also define names to *db* using the *v* command, and the 'clear symbols' command, *cs*, will remove symbols from the memory-resident symbol table.

1.2.1 Code and Data symbols.

db classifies symbols as being either code or data symbols. All symbols in the program's symbol table hunks which occur between the special linker symbols __H1_org__ and __H1_end__ and between __H2_org and __H2_end are considered to be data symbols, and all others are code symbols.

There are two commands for viewing the symbols which are known to *db*: *dc* and *dd*, which display code and data symbols, respectively.

1.2.2 Operator usage of names.

When a C source program is compiled, all global names are prepended with an underscore character. To refer to symbols within *db*, the name can be typed without the underscore. First, the name will be searched for exactly as typed. If not found, *db* will prepend an underscore and search again.

Note that this means to reference a symbol such as *__main*, you would have to type "___main" to differentiate it from the *main* symbol, since typing "__main" would match before the extra underscore is prepended.

1.3 Loading programs and symbols

The *db* program is started independently of the program to be debugged. The *al* command causes the debugger to wait till a program is loaded either from the WorkBench or from the CLI and stops the program before it executes any instructions. The debugger will also load the symbols from the program at this time if the program is in the current directory. If not in the current directory, the 'load symbols' command, *ls*, can be used. 'load symbols' command, *ls*, can be used.

Unfortunately, the Amiga operating system contains no mechanism for terminating a program from an external process. So, there are two ways for the debugger to terminate the program. The first method is to simply allow the program to resume using the *ar* command and let it terminate normally. If the program is unlikely to terminate normally on its own, the *ak* command checks the symbol table for the *__abort* symbol and then the *exit* symbol and sets the program counter to the first one found and allows the program to resume.

1.4 Breakpoints

Before transferring control of the processor to a user's program in response to a *g* command, *db* can set "breakpoints" at specified

locations in the code. When the user's program reaches a breakpoint, *db* regains control.

A breakpoint has a 'skip count' associated with it, which allows a breakpoint to be passed several times before actually taking the breakpoint and returning control to *db* and the user. When a breakpoint is reached, *db* is always activated; it increments a counter associated with the breakpoint. When the counter's value is greater than the breakpoint's skip count, the breakpoint is taken; that is, *db* retains control of the processor. Otherwise, *db* returns control of the processor to the user's program after the breakpoint. By default, a breakpoint's skip count is 0; thus, each time the breakpoint is reached, it's taken.

A breakpoint can also have a sequence of *db* commands associated with it. When a breakpoint is taken, these commands will be executed before *db* allows the operator to enter commands. For example, if you just want to examine a variable each time a certain location in the code is reached and then have the program continue execution, you could define a breakpoint at the location, and specify a list of commands to do just that: the first command in the sequence would be a *d* command to display memory, and the second would be a *g* command to continue execution of the program.

There are two ways to define breakpoints: with the *g* command, and with special breakpoint commands, whose first letter is *b*.

The breakpoint commands manipulate a table of breakpoints: there are commands for entering breakpoints into the table, displaying the entries, reseting their counters, and removing them from the table.

There's a difference between a breakpoint defined in a *g* command and those in the breakpoint table: the *g* command breakpoint is temporary, while a table breakpoint is more permanent (it exists until removed from the table). Before transferring control to the user's program in response to a *g* command, *db* sets all breakpoints that are in the breakpoint table and that are specified in the *g* command itself. When a breakpoint is taken, *db* removes all breakpoints from the code and forgets all about the *g* command breakpoint. The table breakpoints, however, are still in the table and will be set back in memory when control is again returned to the user's program.

db remembers the skip counter associated with a breakpoint which is in the breakpoint table: when it sets breakpoints in memory, the count for such a breakpoint is set to its remembered value (that is, its value in the table); and when a breakpoint is taken, the accumulated count for the breakpoints in memory are saved in the breakpoint table.

1.5 Memory-change breakpoints

The breakpoints described above are taken when a program reaches a specified point in the code. A second type of breakpoint, called a

memory-change breakpoint, is taken when a specified memory location is changed from or set to a particular value.

With a memory-change breakpoint set, *db* will detect either the function or the instruction which modifies the specified memory location, depending on whether the user's program was activated using a *g* command or is being single-stepped using an *s* or *t* command, respectively.

When the user's program is activated with a *g* command and a memory-change breakpoint is set, *db* will examine the specified memory location on entry to, and exit from, each function. It will take a breakpoint, that is, interrupt execution of the program and return control to the operator, when the contents of the memory location meets the specified condition.

When an *s* or *t* command is used to single-step a program and a memory-change breakpoint is set, *db* will examine the specified memory location after each instruction is executed, and take a breakpoint when appropriate.

The *bb* and *bw* commands are used to set and remove memory-change breakpoints.

1.6 Trace mode

db supports a 'trace mode', which displays information whenever a function is entered or exited.

With this mode enabled, on entry to a function, the function or trap name and its arguments are displayed, and, optionally, on exit from a function, its return value is displayed.

The commands *bt* and *bT* affect trace mode: *bt* enables and disables trace mode, and *bT* enables and disables the display of function exit information.

1.7 Backtracing

When *db* regains control from an executing program (for example, because a breakpoint was taken), it has the ability to display information on how the program got to its current location: the *ds* command will display information about the currently executing function, and the function which called it, and so on, back to the Manx function *__main*, which called the user's function *main*.

ds displays, for each function, its name, arguments which were passed to it, and the address to which it will return.

1.8 Macros

db allows the user to define and execute 'macros'; that is, a sequence of *db* commands.

A macro is associated with a single alphabetical character, so up to 26 macros can be known to *db* at any time.

The *db* command *x* is used both to define and execute a macro.

1.9 Other features

Some other features of *db* which haven't yet been discussed are:

* The 'evaluate expression' command, =, does just that.
* The 'help' command, *?*, lists commands.
* The 'input radix' command changes the default radix for input and display.

2. Using DB

2.1 Starting DB

db is started with a command of the form:

```
db
```

or by selecting the **DB** icon and clicking from the WorkBench.

When *db* starts, it creates a window with a startup message and then performs several actions. First, if running under V1.2 or later, the window is created without activating it. Next, *db* will look in the current directory for a file with the name *.dbinit*. If found, *db* will open the file and read and execute commands from the file.

When *db* reaches the end of the file it checks to see whether it is waiting for a breakpoint or program load and if so skips the next step. If *db* is ready for a command, it will first look in the *environment* for a variable DBINIT. If found, *db* will open the file defined by this variable and read and execute commands from this file as well. In this way, it is possible to do system wide as well as local presets.

When *db* has finished reading either or both files it again looks to see if it waiting for a breakpoint or a program load. If it is not, then *db* will activate the window and wait for user input for the next command. If it is, then *db* **will not** activate the window.

What does this mean? Mostly, it means that if you were to set "DBINIT=s:.dbinit" and then put the line:

```
al
```

into that file, when you start *db*, it will open the window, read the file and wait for a program to load without activating the window. This means you can type:

```
db
program
```

without having to do *anything* with the mouse. Nice, huh?

2.2 Termination

If a program is a process or CLI program, *db* will set a breakpoint on the return address from the program. If this breakpoint is reached, *db* will print a message that the program has terminated normally and will automatically resume the task.

2.3 Support For Scatter-Loaded Programs

This version of *db* contains support for scatter-loaded programs. Variables which are referenced using absolute addressing are displayed normally. Functions which are accessed through the jump table are displayed in parentheses. However, the names can be used as though not scatter-loaded.

2.4 Support For Segmented Programs (Overlays)

Yes, amazing as it seems we do support breakpoints in overlays. This is not easy, so there are some caveats. First, any reference to a function in another segment will force the segment into memory. For example, saying:

```
u over1
```

where "over1" is a function will force the segment into memory before disassembling.

If breakpoints are set in the permanent breakpoint table, when the *g* command is given, any breakpoints in non-resident segments will force the segments into memory before setting the breakpoint. (Think of it as a reference to that symbol.) Note that if a segment is unloaded by the program the breakpoint is lost even if the segment is reloaded by the program. However, if a segment is unloaded and the debugger regains control, the next *g* command will reload the segment and reset the breakpoint.

So, its not too bad. The only place that's a problem is if you set a breakpoint, the program unloads the segment and then the function gets called, but the breakpoint is no longer there.

2.5 Input/Output

While *db* is displaying information to the screen, the display can be stopped temporarily by typing "^S". Any key will restart the display. Typing "^C" will abort the display.

When *db* is waiting for a breakpoint or program to load, typing any character will cause *db* to stop the current program. When *db* is displaying while running the program (Trace mode, for example) it is best to use "^C" as other characters may be swallowed by the display routines.

2.6 Commands

This section describes in detail the *db* commands. It first defines some terms that are used in the command descriptions. These terms are *expr*, *term*, *addr*, *range*, and *cmdlist*.

2.6.1 Definitions

2.6.1.1 The Definition of EXPR

An EXPR has the following form:

```
TERM [binop TERM ...]
```

That is, an EXPR can be a single TERM or a series of TERMS separated by binary operators. The binary operators are:

+ -	addition
- -	subtraction
* -	multiplication
/ -	division
% -	modulus
& -	bitwise and
| -	bitwise inclusive or
^ -	bitwise exclusive or

All operators have the same precedence, and an unparenthesized EXPR is evaluated left to right. If you want to override the default order of evaluation of an expression, you can parenthesize the relevant parts of the expression.

An EXPR has a 32-bit value. The operators that are applied to the TERMS out of which the EXPR is built affect just this 32-bit value.

2.6.1.2 The Definition of TERM

A TERM always resolves to a numeric value, and can be one of the following:

REGISTER
CONSTANT
-TERM
ADDR
*ADDR
#ADDR
.
@ [function]
(EXPR)

These names are defined in the following paragraphs.

REGISTER

Registers are specified by their standard names; that is, A0, D0, PC and so on. The value of the TERM is the contents of the register.

CONSTANT

A CONSTANT can be a decimal, hexadecimal, or octal number, or a character.

A sequence of digits preceded by '*0x*' is taken to be a hexadecimal number and those preceded by '*0b*' are binary numbers. A sequence of digits with a leading *0o* is taken to be an octal value. Digit strings ending in . are taken to be decimal values. If none of these prefixes or suffixes are present, the radix of the value is taken from the current radix. The default radix is hexadecimal. Note that if the current radix is set to hexadecimal, numbers must start with a digit to distinguish them from symbols (i.e. *fabc* will be taken to be a symbol where *0fabc* will be taken to be a hexidecimal number.) However, if a symbol is not

matched, and all the letters are hex digits, it will be treated as a number. This is useful, but beware of something like:

```
dw foo+a0
```

because a0 will use the value in register a0 and not the value 0xa0.

A character is represented by the character, surrounded by single quotes, as in 'x'. The value of a character constant is its ASCII value.

Certain characters, the single quote ', and the backslash \ may also be defined within the single quotes. These are identified by a leading backslash character, and are:

char	*hex value*	*db notation*
newline	0a	\n
horizontal tab	09	\t
backspace	08	\b
carriage return	0d	\r
form feed	0c	\f
backslash	5c	\
single quote	27	\'
bit pattern	ddd	\ddd

ADDR

A TERM can be an ADDR; that is, a reference to a location in memory. See the definition of ADDR, below, for more details.

***ADDR**

When a TERM consists of a * followed by an ADDR, the value of the TERM is the contents of the 32-bit field referred to by the ADDR. For example,

*VAR	The contents of the VAR field;
*A7	The contents of the 32-bit field in the data segment pointed at by A7;
*SP	The contents of the 32-bit field on the top of the stack;
*(LBL+2)	The contents of the 32-bit field referred to by LBL+2;

Because an ADDR can itself be an EXPR, the *ADDR term may require extra parentheses. For example,

```
*sp+2
```

is equivalent to *(sp+2) and not (*sp)+2. The value of the first interpretation is the contents of the second word on the stack, while the value of the second is two plus the contents of the first word on the stack.

period(.)

The value of a TERM consisting of a period, '.', is the starting address ADDR of the last similar command. For example, if ten bytes of memory were displayed using the *db* command, as in

```
db 0x100,10
```

then '.' would be set to 0x100 for the next db or dw command. If the next db or dw command is

```
dw .
```

the same 10 bytes would be displayed as words.

The '.' has a separate value for the *u* command, for the *db*, *dw*, and *m* commands, and for the *p* command. An *m* command never modifies its associated '.'.

@ [function]

The @ symbol has as its value the return address of the specified function. The function name is optional, and defaults to the current function. The main use for @ is in the *g* command.

For example,

```
g @
```

transfers control to the user's program, and sets a breakpoint at the return address of the current function.

As another example,

```
g @putc
```

transfers control to the user's program. When the function *putc* is reached, a breakpoint will be set at the address to which it will return.

2.6.1.3 The Definition of ADDR

An ADDR defines the address of a location in memory, and has the form:

```
EXPR
```

Here are some examples of ADDR:

```
pc
main+10
.-40
*sp+8                  Reference to location on the stack.
data+*(a6+6)
```

2.6.1.4 The Definition of RANGE

A RANGE defines a block of memory. It has one of the following forms:

ADDR,CNT
ADDR>ADDR
ADDR
,CNT

The form ADDR,CNT specifies the starting address, ADDR, and a number, CNT. CNT is interpreted differently by different commands. For example, the 'disassemble code' command, *u*, will display CNT lines, while the 'display bytes' command, *db*, will display CNT bytes.

The form ADDR>ADDR specifies the starting and ending addresses of the range.

A full range need not be explicitly specified, because *db* remembers the last-used range and will set unspecified RANGE parameters from the remembered values:

* When a RANGE is specified which consists of a single ADDR, the last used CNT is used.
* When a RANGE is specified which consists of ',CNT', the next consecutive address is used, and the remembered count is changed to the new value.
* When nothing is specified as the RANGE, the next consecutive address is used as the starting ADDR, and the CNT is set to the remembered value.

2.6.1.5 The Definition of CMDLIST

A CMDLIST is a list of commands. It consists of a sequence of commands or macros separated by semicolons:

COMMAND [;COMMAND ...]

If a macro is in a CMDLIST, it must be the last command in the list.

2.7 Command descriptions

The following descriptions of debugger commands uses terms and concepts which were presented in the preceding sections.

The commands are listed alphabetically. For an index, see the command summary which follows the descriptions.

2.7.1 The Amiga Commands

add - Display Device List
adi - Display Interrupt List
adl - Display Library List
adp - Display Port List
adr - Display Resource List

Syntax:

add
adi
adl
adp
adr

Description:

These commands display the addresses and names of one of the various lists pointed to by ExecBase in the Amiga.

---

ai - Display Task Information

Syntax:

ai

Description:

This command displays information about a particular task. If the task is a process or has been run from the CLI, additional information besides the task information is displayed.

If a task has already been selected, the *ai* command displays the information for the selected task. Otherwise, a list of tasks is displayed and the debugger prompts for the task to use.

---

ak - Kill The Current Task

Syntax:

ak

Description:

This command is used to terminate the program or task currently being debugged. It does this by transferring control to the *_abort* or *exit* function of the program.

The Amiga operating system has no provisions for tracking memory or resource allocation. As a result, it is not possible to "kill" an executing task from an external one. Thus, if the program being debugged is unable to resume and exit normally,

the system must be rebooted.

The only alternative is to force the program into its *exit* function. This may or may not work depending upon the state of the program when the *ak* command is given. At the least, some memory is likely to be lost.

This command is made available as a shortcut for the equivalent command "rpc=exit", and discretion in its use is advised.

---

al - Debug The Next Program or Task
aL - Debug The Next Task Without Symbols

Syntax:

al
aL

Description:

Since the debugger operates as an independent task, it cannot load programs directly. Instead, it replaces certain vectors in the system and waits for the next program or task to be loaded. The debugger takes control before the task or program begins to execute.

Thus, to debug a program, this command is given and the debugger waits while the user switches to a new window and either types a CLI command or clicks on a WorkBench ICON to start the program.

After the program is loaded and the debugger gets control, the symbols are loaded from the program file unless the *aL* command is used. Under 1.2, the window is automatically activated when the program is loaded.

---

an - Create A New Debugging Window
aN - Create A New Debugging Window With New Symbols

Syntax:

an
aN

Description:

When the debugger is started, it creates its own window for commands and displays. With the *an* command it is possible to create additional windows for debugging more than one task at a time. If the *an* command is used, the new window will share symbols with the window active when the command was given. This is useful when debugging a sub-task that is a part of the

active program. Note that the *al* command will not read symbols if a symbol table already exists.

When the *aN* command, the new window is created with an independent symbol table. This is useful for debugging several programs simultaneously.

---

am - Memory Information

Syntax:

am

Description:

This command displays the amount of free memory in the system.

---

ap - Debug A Crashed Program (Post-mortem)
aP - Debug A Crashed Program Without Symbols

Syntax:

ap

Description:

You're gonna love this one!

When a program run from the CLI or WorkBench crashes because of an address or bus error or a divide by zero, etc. the trap is handled by a special DOS handler that puts up the "Software Failure" requester. Selecting RETRY has no effect, and selecting CANCEL passes control to EXEC's trap handler which displays the GURU. Haven't you always wished that at least you knew where in the program it died? Well, NOW there's a way to find out!

If *db* is running, or if you can start up db from another window, then using the *ap* command, *db* can act as though it was in control the whole time. When you give the command, *db* will ask you to select the task to be debugged. It will also make an informed guess as to which task it is. After selecting the task, *db* will instruct you to select the CANCEL button in the requester.

Do so, and Presto! *db* will have control right at the point of error. At this point, it is possible to examine the state of things to determine why it happened and possibly to even recover by patching or skipping the code in error. Note that this is not always possible, since memory may have been corrupted before the error occurred.

Note also that if multiple requesters are present for different tasks, this will only work if you choose the most recent task that failed AND click the proper requester.

---

aq - Close ALL Windows

Syntax:

aq

Description:

When more than one window is created using the *an* command, individual windows can be closed using the *q* command. The *aq* command will close all the windows and terminate the debugger. When the last window is closed, the debugger terminates.

When a window is closed, if a task was stopped for debugging, the task is allowed to resume.

The *aq* command performs a *q* command for each of the windows that are open.

---

ar - Allow The Current Task To Resume

Syntax:

ar

Description:

When a task has been selected for debugging, it is kept in a stopped state while commands are being given. The *ar* command deselects the task and allows it to resume. The window returns to the state it was in before a task was selected.

---

as - Select A Task To Debug
aS - Select A Task Without Symbols

Syntax:

as
aS

Description:

It is possible for the debugger to debug a task or program that is already running in the system. The *as* command displays a list of all tasks in the system and prompts for the number of the task to debug. That task is stopped and the symbols for that task are loaded from the program file unless the *aS* form of the command is used.

---

at - Display All Tasks

Syntax:

at

Description:

The *at* command displays a list of all tasks in the system along with their priority, the address of the task block, the status and the name.

2.7.2 The Breakpoint Commands

bb - Set Byte Memory-Change Breakpoint
bw - Set Word Memory-Change Breakpoint
bl - Set Long Memory-Change Breakpoint

Syntax:

bb
bw
bl
bb ADDR [== [VAL]]
bb ADDR [!= [VAL]]
bw ADDR [== [VAL]]
bw ADDR [!= [VAL]]
bl ADDR [== [VAL]]
bl ADDR [!= [VAL]]

Description:

These commands are used to set and clear a memory-change breakpoint, with the parameterized versions used to set breakpoints and the parameter-less version to clear them. The *bb* command is used to monitor a one-byte field, the *bw* command to monitor a two-byte (word) field and the *bl* command is used to monitor a four-byte field.

In the parameterized form of the commands, ADDR specifies the field to be monitored.

With the '==' form, the breakpoint will be triggered when the debugger detects that the field is equal to the specified value, VAL.

With the '!=' form, the breakpoint will be triggered when the debugger detects that the field is different from the specified value.

The VAL parameter is optional. If not specified, it defaults to the current value at the ADDR. If the comparison field is also not specified, the default is "!=". Thus, to break when a location is

changed, the command:

bw ADDR

is sufficient.

---

bc - Clear a single breakpoint
bC - Clear all breakpoints

Syntax:

bc ADDR
bC

Description:

These commands delete breakpoints from the breakpoint table.

bc deletes the single breakpoint specified by the address ADDR, and *bC* deletes all breakpoints from the table.

---

bd - Display breakpoints

Syntax:

bd

Description:

bd indicates what special breakpoints are set and displays all entries in the breakpoint table.

For each breakpoint, the following information is displayed:

* Its address, using a symbolic name, if possible.
* The number of times it's been 'hit' without a breakpoint being taken.
* The skip count for it;
* The command list for it, if any.

For example, a *bd* display might be:

address	hits	skip	command
_printf	1	2	
_putc	0	0	db _Cbuffs

In this example, two breakpoints are in the table. The first is at the beginning of the function *_printf*; a breakpoint will be taken for it every third time it is reached, and no command will be executed. Given its current hit count, a breakpoint will be taken the second time *_printf* is reached.

The second breakpoint is at the function *__putc*; a breakpoint will be taken each time the function is reached, and will display memory, in bytes, starting at *__Cbuffsi*.

---

bh - Set Memory Hash Breakpoint

Syntax:

bh [RANGE]

Description:

bh performs a simple checksum on the specified range of memory. Breakpoints are automatically set at the entry and exit of all functions. The checksum is checked each time the debugger regains control of the task. If the checksum is ever detected to be different, the debugger will stop the program at that point.

---

bq - Toggle Low Memory Checksum

Syntax:

bq

Description:

When the debugger starts, it checksums the first 256 bytes of memory. Each time the debugger regains control, this checksum is calculated. If the checksum differs from the saved value, a message is displayed and the program stopped. The new checksum is saved so multiple breakpoints are not generated.

The *bq* command toggles whether the checksum is done or not. Turning off checksumming will speed single stepping since otherwise the checksum is computed for each instruction executed.

---

br - Reset breakpoint counters

Syntax:

br [ADDR]

Description:

br resets the 'hit' counter for the specified breakpoint which is at the address, ADDR. If ADDR isn't given, the 'hit' counters for all breakpoints in the breakpoint table are reset.

---

bs - Set or modify a breakpoint

Syntax:

[#] bs ADDR [;CMDLIST]

Description:

bs enters a breakpoint into the breakpoint table, or modifies an existing entry.

The optional parameter # is the skip count for the breakpoint. If not specified, the skip count is set to 0, meaning that each time the breakpoint is reached it will be taken.

The optional parameter CMDLIST is a list of debugger commands to be executed when the breakpoint is taken.

---

bt - Toggle the trace mode flag
bT - Toggle the return trace mode flag

Syntax:

bt
bT

Description:

bt and *bT* toggle the trace mode and return trace mode flags, respectively.

The state of the trace mode flag determines whether trace mode is enabled or disabled.

The state of the return trace mode flag determines whether the tracing of a function's return is enabled or disabled. If trace mode is disabled, the return trace mode flag has no effect.

Note that recursive calls will not handle the return trace mode correctly.

---

bu - User Defined Breakpoint

Syntax:

bu ADDR
bT

Description:

bu sets the address of a user defined breakpoint function.

This command sets a pointer to a function which is called each time the debugger regains control. The function has no effect when it returns a zero value. When a non-zero value is returned, a message displaying the value is displayed and the task stopped.

The function is called as a C function and is expected to follow the standard C register saving conventions. Passed as an argument to the function is the address of an array of longs which contain the tasks register values, including the program counter and status register. The order of items in the array is:

registers D0-D7
registers A0-A7
status register
program counter

This function can be used to breakpoint on special conditions. For example, the program can be stopped when the D0 and D1 registers contain the same value. The function would look like this:

```
    moveq   #0,d0
    move.l  4(sp),a0
    move.l  (a0),d1
    cmp     4(a0),d1
    bne     1$
    moveq   #1,d0
1$  rts
```

Setting the address to zero disables the breakpoint.

Thanks go to Rick Ross and Richard Williamson for this one!

2.7.3 The Clear Commands

cs - Clear symbol table

Syntax:

cs

Description:

cs removes all symbols from the debugger's memory-resident symbol table.

2.7.4 The Display Commands

db - Display memory in bytes
dw - Display memory in words
dl - Display memory as longs
d - Display memory in last format

Syntax:

db [RANGE]
dw [RANGE]
dl [RANGE]
d [RANGE]

Description:

The *db*, *dw* and *dl* commands display successive bytes, words and double words of memory, respectively. *d* displays memory using the last format specified; for example, if *d* is entered, and *db* was the last 'display memory' command, then *d* will display bytes, too.

The starting address of the RANGE parameter is optional; if not specified, it defaults to the ending address of the last display's RANGE, plus one.

Each line of the display begins with the address, followed by a hexadecimal display of 16 bytes, 8 words or 4 double words, followed by an ASCII display, by bytes, of the same data. For the ASCII display, values falling outside the range 0x20 to 0x7f are displayed as a period.

If the ending address does not fall on a multiple of 16 bytes, only the number of bytes or words specified in the last line will be displayed.

---

dc - Display all code symbols

Syntax:

dc

Description:

dc lists all the code symbols in the memory-resident symbol table and all user-defined symbols.

For each symbol, its name and address are displayed.

---

dd - Display all data symbols

Syntax:

dd

Description:

dd lists all the data symbols in the memory-resident symbol table.

For each symbol, its name and address are displayed.

---

dg - Display global values

Syntax:

dg

Description:

For each data symbol in the debugger's symbol table, *dg* displays the contents of the 16-bit field referenced by that symbol.

---

ds - Display Stack Backtrace

Syntax:

ds

Description:

ds displays information about the current function, the function which called it, and so on, back to *__main*, the Manx function which called the user's function *main*.

For each function, the information consists of the function's name, the parameters passed to it, and the address to which it will return.

The arguments are displayed as a series of 16-bit hex values. If an argument is actually of type long or double, it will still be displayed as separate words.

ds determines the number of parameters by looking at the instructions which follow the address to which the function will return.

ds assumes that the A5 register points to the C stack frame for the current function, unless the current instruction is within 4 bytes of the start of the function.

2.7.5 The Go commands

g - Execute the program
G - Execute the program, without setting table breakpoints

Syntax:

[#]g [@ <function>] [ADDR] [;CMDLIST]
[#]G [@ <function>] [ADDR] [;CMDLIST]

Description:

The *g* commands transfer control of the processor to the user's program, at the address specified by PC. The user's program then executes until it terminates, an error such as division by zero occurs, or a breakpoint is taken; control then returns to the debugger program.

The parameters to the 'g' commands allow one or two temporary breakpoints to be set in memory before the user's program is executed.

The difference between the 'g' and the 'G' command is that the 'G' command sets in memory just the breakpoints specified in the command itself, while the 'g' command also sets the breakpoints specified in the breakpoint table.

The '#' and 'ADDR' parameters define one of the temporary breakpoints that a Go command can set:

* # is the skip count for the breakpoint; it defaults to zero, meaning that the breakpoint is taken every time it's reached;
* ADDR is the address for the breakpoint;

The '@ <function>' parameter specifies that a temporary breakpoint is to be set at the return address of the specified function. If the function isn't specified, it defaults to the current function. If a function is specified, the breakpoint is set to the address to which the function will return. In this case, the breakpoint isn't set until the function is entered; thus, in programs which call the function from several different places, the breakpoint will be set at the actual address to which the function will return.

The ';CMDLIST' parameter defines a sequence of debugger commands, separated by semicolons, that the debugger is to execute once a breakpoint which is specified in the 'go' command is taken. If this parameter isn't specified, it defaults to the command list used for the last temporary breakpoint.

Before setting breakpoints and transferring control to the user's program, the debugger single-steps the user's program, (that is,

causes it to execute one instruction). This allows the operator to transfer control to a location in the program at which there is a breakpoint, without immediately triggering a breakpoint and re-entry to the debugger.

2.7.6 The Load Commands

ls - load symbols

Syntax:

ls progfile

Description:

ls loads symbols from the specified program file into the debugger's memory-resident symbol table, after first clearing the memory-resident table of all but those symbols defined with the *v* command.

2.7.7 The Memory Modification Commands

ma - Allocate Some Memory

Syntax:

ma VAL

Description:

ma allocates VAL bytes of memory and displays the address of the memory.

This command is mostly used to allocate memory for patches and possibly to hand assemble a small user break routine. The "." associated with memory change and display is set to the address as well making it possible to load patches or user functions from a file.

---

mb - Modify bytes of memory
mw - Modify words of memory
ml - Modify double words of memory

Syntax:

mb ADDR EXPR1 [EXPR2 ...]
mw ADDR EXPR1 [EXPR2 ...]
ml ADDR EXPR1 [EXPR2 ...]

Description:

mb, *mw* and *ml* modify bytes and words of memory, respectively.

The parameter ADDR specifies the address of the first byte or word to be modified.

The EXPR parameters are expressions, whose resulting values are set in memory, with EXPR1 set in the first byte or word specified, EXPR2 set in the next higher byte or word, and so on.

The EXPR parameters can be separated by spaces or commas.

---

mc - Compare memory

Syntax:

mc RANGE = ADDR

Description:

mc compares two blocks of memory and, for each comparison which fails, displays the corresponding address, and value.

RANGE specifies one of the blocks of memory. The second begins at ADDR and has the same length as the first block.

---

mf - Fill memory

Syntax:

mf RANGE = EXPR

Description:

mf sets each byte in a block of memory to a specified value.

The RANGE parameter specifies the memory block, and EXPR an expression whose resulting value is the value to be set in the range.

---

mm - Move memory

Syntax:

mm RANGE = ADDR

Description:

mm copies one block of memory to another.

The RANGE parameter specifies the source block and ADDR the starting address of the block to be modified.

---

ms - Search memory

Syntax:

ms RANGE = EXPR1 [EXPR2 ...]

Description:

ms searches a block of memory for a sequence of bytes having specified values. For each match, the corresponding address of the start of the string is displayed.

RANGE specifies the block of memory. The EXPR parameters are expressions, each of whose resulting values is one byte of the search sequence.

2.7.8 The Radix Command

n - Change radix

Syntax:

nX
n

Description:

This command changes the default radix (hexadecimal) for user input or debugger display. X is a single character 'b', 'd', 'o', or 'x' which changes the default radix to binary, decimal, octal, or hexadecimal respectively. If the X is omitted, a message giving the current radix is displayed.

The radix can be forced to a given value by specifying one of the following prefixes before the desired number:

0x	hex
0o	octal
0b	binary

To display a number in decimal, the number must have a . (period) appended to it.

2.7.9 The 'Print' Command

p - formatted print

Format:

p[format] [ADDR][,COUNT]

Description:

p generates a formatted display of memory of a section of memory, by converting data items in memory to a displayable form as directed by the format conversion string *format*.

format is a list of format specifications, each of which defines the type of a data item and the conversion to be performed on it.

p works its way through the *format* string, converting and displaying data items in memory as requested by the format string items. When *p* reaches an item in the *format* string, it converts the data item at its 'current address' as directed by the format item. When it finishes processing a format string item, it increments its current address by the size of the data item that it just processed, so as to be ready to process the next data item as directed by the next format string item.

The *format* string is optional; if not specified, the *format* string used by the previous *p* command is used.

ADDR specifies the address of the first data item that *p* is to convert and display. If ADDR is not entered, the starting address is assumed to be the print command's 'current address'. Normally, this is the address of the first byte beyond the last data item converted by the last *p* command. However, there is a *format* item that causes *p* to remember the address contained in the current data item, and then make that the current address after it finishes processing the entire format string.

COUNT specifies the number of times that *p* is to work its way through the *format* string. Each time through, *p* begins at the current address that was left by the last time through. If COUNT isn't specified, it defaults to one time.

The format items have the form

[rpt][indir_flgs][size]desc_code

where

* *desc_code* is a single-letter code that defines the type of the data item and the conversion to be performed upon it. For example, the code *d* says 'take the two-byte binary value at the current address, convert it to decimal, and print it'. So if *var* is an *int*, the following command could be used to print its value in decimal:

  pd var

  The code *x* says "take the two-byte binary value at the current address, convert it to hexadecimal, and print the result". So the hexadecimal value of *var* could be printed with the command:

```
px var
```

* *indir* is a string of zero or more * characters, which are indirection indicators specifying that the value at the current data item is a pointer to a chain of zero or more pointers, the last of which points to an object whose type and requested conversion are defined by *desc_code*.

  To find the data object corresponding to a format item that has indirection indicators, *p* begins by setting its idea of the address of the data object to the current address. It then works its way from left to right through the indirection indicators; for each indicator it replaces its current idea of the data object address with the pointer that is in the field at this address. The data object address is distinct from the current address: at the end of this process, the *p* command's current address is simply incremented past the first pointer.

  A * specifies that the pointer within the field referenced by the current data object address is four bytes long. This pointer is the offset component of the new data object address from the last segment referenced.

  For example, if the variable *cp* is a pointer to a character string (that is, its declaration is *char *cp*), then the string pointed at by *cp* could be printed by the command

```
p*s cp
```

  Here we have made use of the *s desc_code*, which specifies that the data object is a character string, and that the string's characters are to be printed, with possible modifications as noted below, up to a terminating null character. After this command, the *p* command's current address is set to the byte immediately following *cp*.

  As another example, if *cpp* is a pointer to an array of pointers to character strings (that is, the declaration of *cpp* is *char **cpp*), then the string pointed at by the first element of the array could be displayed with the command

```
p**s cpp
```

  Following this command, the *p* command's current address is set to the byte following *cpp*.

* The *rpt* parameter of a format item defines the number of times that the item is to be processed. It allows a sequence of *rpt* identical format items to be abbreviated by just one such item with a leading *rpt* count.

For example, if *a* is an array of *float*s, then the first five items in this array could be displayed with the command

```
p5f a
```

This command uses the fact that the *desc_code* to convert a four-byte floating point value at the current address to a displayable value is *f*. This command is equivalent to the command *pfffff a*. At the end of this command, the *p* command's current address is set to the address of the byte following the last displayed *float*.

* The *size* parameter of a format item defines the number of data items that are to be converted and printed. When the format item doesn't use indirection, *size* has the same effect as *rpt*; for example, in the *p5f a* command above, the 5 could be interpreted as being a *size* parameter instead of a *rpt* parameter.

  When the format item does use indirection, then the *size* parameter defines the number of data items to be converted and printed at the end of the indirection chain. For example, if a module defines *lpp* as a pointer to an array of pointers to array of *long*s (that is, the declaration of *lpp* is *long **ip*), then the following command would display the first four *long*s pointed at by the first element of the pointer array:

```
p**4D lpp
```

  Here we have used the *D desc_code*, which specifies that a four-byte signed binary value is to converted to decimal and printed. The following command would display the first three *long*s pointed at by the first three elements of the pointer array:

```
p3*4D *lpp
```

To demonstrate further the difference between the *rpt* and *size* fields in a format item, consider the format items *4*d* and **4d*. The first causes the print command to take the item at the current address as a pointer, increment the current address by four, convert to decimal and print the two-byte value referenced by the pointer, and then repeat the process three more times. At the end of the process, the current address has been advanced by sixteen.

The second item causes the print command to again take the item at the current address as a pointer, increment the current address by four, and then convert to decimal and print the four successive two-byte values that begin at the address defined by the pointer. At the end of the process, the current address has

been advanced by four.

As an example of the use of format strings containing several format items, consider the following code in a program that uses short data pointers:

```
struct {
    int *ip;
    float flt;
    char *cp;
} var = {&i, 3.14159, "ralph");
int i=2;
```

The command

```
p*d2-xf*s2-x var
```

will print

```
2 xxxx 3.14159 ralph yyyy
```

where *xxxx* is the hexadecimal address of *i* and *yyyy* is the hexadecimal address of the string.

A complete list of the *desc_codes*

We have introduced some of the *desc_codes* above. Here is a list of the basic *desc_codes*:

b	Convert to hexadecimal and print a byte.
d	Convert to decimal and print a two-byte signed binary value.
D	Convert to decimal and print a four-byte signed binary value.
f	Convert and print a four-byte *float.*
F	Convert and print an eight-byte *double.*
o	Convert to octal and print a two-byte field.
O	Convert to octal and print a four-byte field.
x	Convert to hexadecimal and print a 2-byte field.
X	Convert to hexadecimal and print a 4-byte field.
u	Convert to decimal and print an unsigned, two-byte value.
U	Convert to decimal and print an unsigned, four-byte value.
p	Print a pointer in address form with translation.
P	Print a pointer in address form without translation.
c	Print a character with translation.
C	Print a character without translation.
s	Print a string up to a terminating null byte with translation.
S	Print a string up to a terminating null byte without translation.

For the C and S *desc_codes*, each character is printed "as is", with no translations.

For the c and s codes, printable ASCII characters (that is, whose hex value is between 0x20 and 0x7f) are printed "as is". A character whose hex value is less than 0x20 is printed as two characters: ^ followed by the printable character whose hex value equals the original character's value plus 0x40. A character whose hex value is 0x80 or greater is displayed as a ' character followed by the one or two characters that would be printed for the character whose hex value equals that of the original character less 0x80. For example, 0x41, 0x1, and 0x81 would be printed as A, ^A, and '^A, respectively.

The following *desc_codes* can be used to assist in the formatting of the *p* output:

character	*output*
N or n	Output a newline character
R or r	Output a blank character
T or t	Output a tab character
"string"	output "string"

These characters can be preceded by a count specifying the number of characters or strings to be output.

The next group of *desc_codes* change the *p* command's notion of the current address. They don't cause any printing.

^ Back up the current address by the size of the last data item.

- or + Back up or advance, respectively, the current address by *size* bytes, where *size* is a decimal value preceding the - code. If *size* isn't specified, it defaults to one byte.

A or a Remember the pointer that is contained in the current data object; If this pointer is not null, set the *p* command's current address to this value after the entire format string has been processed.

If the pointer is null, set the *p* command's current address to the value it had before the entire format string was processed.

The A and a *desc_codes* are useful for printing the elements of a linked list. For example, consider the following code, which defines the structure for a symbol table item, and declares *sym_head* to be a pointer to this structure. The program that uses this structure and field will chain symbol table items together, and set a pointer to the head of the chain in *sym_head*.

```
struct symbol {
   struct symbol * sym_next;
   char *sym_name;
   unsigned sym_val;
} *sym_head;
```

The following command would display the symbol table item pointed at by *sym_head* and then set the *p* command's current address to the next symbol table item, which is pointed at by the *sym_next* field in the first item:

```
pA"symbol name="*snt"value="x sym_head
```

After this command is entered, you can display successive symbol table items by simply entering

```
p
```

The *p* command's current address is correctly set to the next table item, and since a format string isn't specified, the *p* command will use the one that it last used.

You can print out multiple symbol table items by entering a single *p* command. To do this, place a comma and the maximum number of items to be printed after the command's starting address. The command will follow the chain, printing symbol table items until it either prints the specified number of items or it prints an item whose *sym_next* pointer is null. In the latter case, it will terminate and leave the *p* command's current address set to the address of the last symbol table item. For example, entering

```
pA"symbol name="*snt"value="x sym_head,100
```

will print symbol table items until it either prints 100 items or it prints an item having a null *sym_next* pointer.

2.7.10 The Quit command

q - Quit the debugger

Syntax:

q

Description:

When more than one window is created using the *an* command, individual windows can be closed using the *q* command. When the last window is closed, the debugger terminates.

When a window is closed, if a task was stopped for debugging, the task is allowed to resume.

2.7.11 The Register command

r - Register display

Syntax:

r
r <reg>=EXPR

Description:

r displays and modifies the registers, including the status registers, of the program being debugged.

The parameter-less version displays the registers. If a 68881 is in use for this task, it's registers are displayed as well, though they may not be modified.

The parameterized version modifies the contents of a register, with <reg> being the name of the register to be modified, and EXPR an expression whose resulting value is to be set into the register.

2.7.12 The Single Step commands

s - Single step with display
S - Single step without display
t - Single step with display over subroutines
T - Single step without display over subroutines

Syntax:

[#] s [;CMDLIST]
[#] S [;CMDLIST]
[#] t [;CMDLIST]
[#] T [;CMDLIST]

Description:

These commands 'single step' the user's program; that is, execute its instructions one by one. The 's' versions of the command single step through each and every instruction. The 't' versions allow the user to treat *jsr* and *bsr* instructions specially. The debugger does not get control until the routine called by the instruction returns.

The optional '#' parameter specifies the number of instructions to be executed; it defaults to one instruction. However, a value of 0 specifies a continuous mode which will only stop by user intervention or when one of the other breakpoints is encountered.

The optional CMDLIST parameter is a list of debugger commands to be executed after each single step.

The commands differ in that *s* and *t* display information after each single step, whereas *S* and *T* only display information after the last single step.

The displayed information consists of the registers and a disassembly of the next instruction to be executed.

2.7.13 The Unassemble commands

u - Unassemble memory, with symbols
U - Unassemble memory, without symbols

Syntax:

u RANGE
U RANGE

Description:

These commands 'disassemble' a range of memory; that is, display the assembly language instructions in the range.

The *u* and *U* commands differ in that the *u* command will make use of the symbol table during disassembly and the *U* command won't. Also, the *U* command displays, for each instruction, the hex value of each byte of the instruction, whereas the *u* command won't.

With the *u* command, the disassembly of an instruction which references memory displays the location as the symbol nearest to the location plus an offset, if possible. With the *U* command, the location is displayed as a hexadecimal value.

The RANGE parameter specifies the area of memory to be disassembled. It gives the starting address, and either the number of instructions to be disassembled, or the ending address of the area.

2.7.14 The Variable commands

v - Create a new symbol
V - Modify the value of an existing symbol

Syntax:

v SYMBOL = ADDR
V SYMBOL = ADDR

Description:

The *v* and *V* commands are used to create a new symbol or modify the value for an existing symbol, respectively, in the debugger's memory resident symbol table.

SYMBOL is the name of the symbol being created or modified, and ADDR is its address.

The symbol will be classified as a code symbol.

2.7.15 The Macro command

x - Macro command

Syntax:

xc
xc = CMDLIST
x?

Description:

The *x* command defines or executes a sequence of debugger commands, called a 'macro'. It can also list the defined macros.

A macro is associated with a letter of the alphabet, so up to 26 macros can be known to the debugger at one time. Case is not significant.

A macro is defined by typing the letter 'x', followed by the letter with which the macro is to be associated. Then follows an '=' character and the macro's list of debugger commands, with the commands separated by semicolons.

A macro is executed by typing 'x', followed by the letter with which the macro is associated, followed by a carriage return.

The macros which have been defined can be listed using the command *x?*. The output format is suitable for reading back in from a file.

2.7.16 The 'Display expression' Command

= - Display the value of an expression

Syntax:

= EXPR

Description:

This command displays the value of an expression.

The expression is displayed in several formats: hexadecimal, signed decimal, unsigned decimal, octal, binary, and ASCII. If a symbol table has been loaded, the closest symbol is displayed as well.

2.7.17 The Redirect Input/Output Commands

< - Take Input From File
> - Log Output To File
>> - Log Commands Only

Syntax:

< file
> file
>> file

Description:

These commands are used to temporarily redirect input and output to the files specified.

The < command causes the input which normally comes from the keyboard to be taken from the named file instead. This continues until the end of file is reached. This is especially useful for defining macros.

The > command causes the output of all commands to be displayed in the window and also written to the file specified. This continues until either a different file is specified using the > command or until the > command is used with no file name. In this case, the current file is closed and redirection stops.

The >> command causes a log of the commands only to be saved in a file. This is useful if a complicated set of commands is needed to get a program to a certain point for debugging. After the commads have been logged to the file, the file can be used as input via the < command to retrace the steps to reach the same point.

2.7.18 The Help command

? - list commands

Syntax:

?

Description:

This command lists the debugger commands. For groups of related commands, the listing usually lists the first letter of the commands followed by a ?. You can get a listing of all the commands in such a group by typing the the letter, the ?, and return. For example, the listing for the 'display' commands is *d?*; thus you can type *d?* followed by return to get a listing of all the 'display' commands.

3. Command Summary

amiga commands

add	display device list
adi	display interrupt list
adl	display library list
adp	display port list
adr	display resource list
ai	display task information
ak	kill the current task
al/aL	wait for next task load with/without symbols
am	display memory information
an	make a new debug window
ap/aP	select task for post-mortem
aq	close all windows
ar	release current task
as/aS	select task to stop with/without symbols
at	display task list

breakpoint commands

bb/bw/bl	set byte/word/long memory-change breakpoint
bc/bC	clear one/all breakpoints
bd	display the breakpoint table
bh	set checksum breakpoint
bq	toggle low memory checksum
br	reset the breakpoint counters
bs	set or modify a breakpoint
bt/bT	enable/disable trace mode
bu	set user defined breakpoint

clear commands

cs	clear all symbols

display commands

db/dw/dl/d	display memory in bytes/words/longs/last format
dc/dd	display code/data symbols
dg	display global values
ds	display stack backtrace

go commands

g/G	execute user's program

load commands

ls	load symbols

memory modification commands

ma	allocate some memory
mb/mw/ml	modify bytes/words/longs of memory
mc	compare areas of memory
mf	fill memory
mm	move memory
ms	search memory

radix command

n	change the default radix for input and display

formatted print commands

p	generate formatted print

quit command

q	close current window

register command

r	register display

single step commands

s/S	single step with/without display
t/T	single step with/without display over calls

unassembly commands

u/U	unassemble memory

variable command

v/V	create/modify symbol

macro command

x	define or modify a command macro

display expression

=	display value of an expression

input/output commands

<	take input from file
>	log output to file
>>	log commands to file

help command

?	list debugger commands

OVERVIEW OF LIBRARY FUNCTIONS

Chapter Contents

Overview of Library Functions ..... libov
1. I/O Overview 4
1.1 Pre-opened devices, command line args 4
1.2 File I/O 6
1.2.1 Sequential I/O 6
1.2.2 Random I/O 6
1.2.3 Opening Files 6
1.3 Device I/O 7
1.3.1 Console I/O 7
1.3.2 I/O to Other Devices 7
1.4 Mixing unbuffered and standard I/O calls 7
2. Standard I/O Overview 9
2.1 Opening files and devices 9
2.2 Closing Streams 9
2.3 Sequential I/O 10
2.4 Random I/O 10
2.5 Buffering 10
2.6 Errors 11
2.7 The standard I/O functions 12
3. Unbuffered I/O Overview 14
3.1 File I/O 15
3.2 Device I/O 15
3.2.1 Unbuffered I/O to the Console 15
3.2.2 Unbuffered I/O to Non-Console Devices 16
4. Console I/O Overview 17
4.1 Line-oriented input 17
4.2 Character-oriented input 18
4.3 Using ioctl 19
4.4 The sgtty fields 19
4.5 Examples 20
5. Dynamic Buffer Allocation 22
6. Error Processing Overview 23

Overview of Library Functions

This chapter presents an overview of the functions that are provided with Aztec C. It's divided into the following sections:

1. *I/O:* Introduces the i/o system provided in the Aztec C package.

2. *Standard I/O:* The i/o functions can be grouped into two sets; this section describes one of them, the standard i/o functions.

3. *Unbuffered I/O:* Describes the other set of i/o functions, the unbuffered.

4. *Console I/O:* Describes special topics relating to console i/o.

5. *Dynamic Buffer Allocation:* Discusses topics related to dynamic memory allocation.

6. *Errors:* Presents an overview of error processing.

The overviews present information that is system independent. Overview information that is specific to your system is in the form of an appendix to this chapter; it accompanies the system dependent section of your manual.

1. Overview of I/O

There are two sets of functions for accessing files and devices: the unbuffered i/o functions and the standard i/o functions. These functions are identical to their UNIX equivalents, and are described in chapters 7 and 8 of *The C Programming Language*.

The unbuffered i/o functions are so called because, with few exceptions, they transfer information directly between a program and a file or device. By contrast, the standard i/o functions maintain buffers through which data must pass on its journey between a program and a disk file.

The unbuffered i/o functions are used by programs which perform their own blocking and deblocking of disk files. The standard i/o functions are used by programs which need to access files but don't want to be bothered with the details of blocking and deblocking the file records.

The unbuffered and standard i/o functions each have their own overview section (UNBUFFERED I/O and STANDARD I/O). The remainder of this section discusses features which the two sets of functions have in common.

The basic procedure for accessing files and devices is the same for both standard and unbuffered i/o: the device or file must first be "opened", that is, prepared for processing; then i/o operations occur; then the device or file is "closed".

There is a limit on the number of files and devices that can simultaneously be open; the limit on your system is defined in this chapter's system dependent appendix.

Each set of functions has its own functions for performing these operations. For example, each set has its own functions for opening a file or device. Once a file or device has been opened, it can be accessed only by functions in the same set as the function which performed the open, and must be closed by the appropriate function in the same set. There are exceptions to this non-intermingling which are described below.

There are two ways a file or device can be opened: first, the program can explicitly open it by issuing a function call. Second, it can be associated with one of the logical devices standard input, standard output, or standard error, and then opened when the program starts.

1.1 Pre-opened devices and command line arguments

There are three logical devices which are automatically opened when a program is started: standard input, standard output, and standard error. By default, these are associated with the console. The operator, as part of the command line which starts the program, can specify that these logical devices are to be "redirected" to another

device or file. Standard input is redirected by entering on the command line, after the program name, the name of the file or device, preceded by the character '<'. Standard output is redirected by entering the name of the file or device, preceded by '>'.

For example, suppose the executable program *cpy* reads standard input and writes it to standard output. Then the following command will read lines from the keyboard and write them to the display:

```
cpy
```

The following will read from the keyboard and write it to the file *testfile*:

```
cpy >testfile
```

This will copy the file *exmplfil* to the console:

```
cpy <exmplfil
```

And this will copy *exmplfil* to *testfile*:

```
cpy <exmplfil >testfile
```

Aztec C will pass command line arguments to the user's program via the user's function *main(argc, argv)*. *argc* is an integer containing the number of arguments plus one; *argv* is a pointer to a an array of character pointers, each of which, except the first, points to a command line argument. On some systems, the first array element points to the command name; on others, it is a null pointer. Information on your system's treatment of this pointer is presented in this chapter's system dependent appendix.

For example, if the following command is entered:

```
prog arg1 arg2 arg3
```

the program *prog* will be activated and execution begins at the user's function *main*. The first parameter to main is the integer 4. The second parameter is a pointer to an array of four character pointers; on some systems the first array element will point to the string "prog" and on others it will be a null pointer. The second, third, and fourth array elements will be pointers to the strings "arg1", "arg2", and "arg3" respectively.

The command line can contain both arguments to be passed to the user's program and i/o redirection specifications. The i/o redirection strings won't be passed to the user's program, and can appear anywhere on the command line after the command name. For example, the standard output of the "prog" program can be redirected to the file outfile by any of the following commands; in each case the argc and argv parameters to the main function of 'prog' are the same as if the redirection specifier wasn't present:

```
prog arg1 arg2 arg3 >outfile
prog >outfile arg1 arg2 arg3
prog arg1 >outfile arg2 arg3
```

1.2 File I/O

A program can access files both sequentially and randomly, as discussed in the following paragraphs.

1.2.1 Sequential I/O

For sequential access, a program simply issues any of the various read or write calls. The transfer will begin at the file's "current position", and will leave the current position set to the byte following the last byte transferred. A file can be opened for read or write access; in this case, its current position is initially the first byte in the file. A file can also be opened for append access; in this case its current position is initially the end of the file.

On systems which don't keep track of the last character written to a file, it isn't always possible to correctly position a file to which data is to be appended. If this is a problem on your system, it's discussed in the system dependent appendix to this chapter, which accompanies the system dependent section of your manual.

1.2.2 Random I/O

Two functions are provided which allow a program to set the current position of an open file: *fseek*, for a file opened for standard i/o; and *lseek*, for a file opened for unbuffered i/o.

A program accesses a file randomly by first modifying the file's current position using one of the seek functions. Then the program issues any of the various read and write calls, which sequentially access the file.

A file can be positioned relative to its beginning, current position, or end. Positioning relative to the beginning and current position is always correctly done. For systems which don't keep track of the last character written to a file, positioning relative to the end of a file can't always be correctly done. For information on this, see this chapter's system dependent appendix.

1.2.3 Opening files

Opening files is somewhat system dependent: the parameters to the open functions are the same on the Aztec C packages for all systems, but some system dependencies exist, to conform with the system conventions. For example, the syntax of file names and the areas searched for files differ from system to system.

For information on the opening of files on your system, see this chapter's system dependent appendix.

1.3 Device I/O

Aztec C allows programs to access devices as well as files. Each system has its own names for devices: for the names of devices on your system, see this chapter's system dependent appendix.

1.3.1 Console I/O

Console I/O can be performed in a variety of ways. There's a default mode, and other modes can be selected by calling the function *ioctl*. We'll briefly describe console I/O in this section; for more details, see the *Console I/O* section of this chapter and the system dependent appendix to this chapter.

When the console is in default mode, console input is buffered and is read from the keyboard a line at a time. Typed characters are echoed to the screen and the operator can use the standard operating system line editing facilities. A program doesn't have to read an entire line at a time (although the system software does this when reading keyboard input into it's internal buffer), but at most one line will be returned to the program for a single read request.

The other modes of console i/o allow a program to get characters from the keyboard as they are typed, with or without their being echoed to the display; to disable normal system line editing facilities; and to terminate a read request if a key isn't depressed within a certain interval.

Output to the console is always unbuffered: characters go directly from a program to the display. The only choice concerns translation of the newline character; by default, this is translated into a carriage return, line feed sequence.

Optionally, this translation can be disabled.

1.3.2 I/O to Other Devices

On most systems, few options are available when writing to devices other than the console. For a discussion of such options, if any, that are available on your system, see this chapter's system dependent appendix.

1.4 Mixing unbuffered and standard i/o calls

As mentioned above, a program generally accesses a file or device using functions from one set of functions or the other, but not both.

However, there are functions which facilitate this dual access: if a file or device is opened for standard i/o, the function *fileno* returns a file descriptor which can be used for unbuffered access to the file or device. If a file or device is open for unbuffered i/o, the function *fdopen* will prepare it for standard i/o as well.

Care is warranted when accessing devices and files with both standard and unbuffered i/o functions.

2. Overview of Standard I/O

The standard i/o functions are used by programs to access files and devices. They are compatible with their UNIX counterparts, with few exceptions, and are also described in chapter 8 of *The C Programming Language*. The exceptions concern appending data to files and positioning files relative to their end, and are discussed below.

These functions provide programs with convenient and efficient access to files and devices. When accessing files, the functions buffer the file data; that is, handle the blocking and deblocking of file data. Thus the user's program can concentrate on its own concerns.

Buffering of data to devices when using the standard i/o functions is discussed below.

For programs which perform their own file buffering, another set of functions are provided. These are described in the section UNBUFFERED I/O.

2.1 Opening files and devices

Before a program can access a file or device, it must be "opened", and when processing on it is done it must be "closed".

An open device or file is called a "stream" and has associated with it a pointer, called a "file pointer", to a structure of type FILE. This identifies the file or device when standard i/o functions are called to access it.

There are two ways for a file or device to be opened for standard i/o: first, the program can explicitly open it, by calling one of the functions *fopen*, *freopen*, or *fdopen*. In this case, the open function returns the file pointer associated with the file or device. *fopen* just opens the file or device. *freopen* reopens an open stream to another file or device; it's mainly used to change the file or device associated with one of the logical devices standard output, standard input, or standard error. *fdopen* opens for standard i/o a file or device already opened for unbuffered i/o.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file pointer is *stdin*, *stdout*, or *stderr*, respectively. These symbols are defined in the header file *stdio.h*. See the section entitled I/O for more information on logical devices.

2.2 Closing streams

A file or device opened for standard i/o can be closed in two ways: first, the program can explicitly close it by calling the function *fclose*.

Alternatively, when the program terminates, either by falling off the end of the function *main*, or by calling the function *exit*, the system will automatically close all open streams.

Letting the system automatically close open streams is error-prone: data written to files using the standard i/o functions is buffered in memory, and a buffer isn't written to the file until it's full or the file is closed. Most likely, when a program finishes writing to a file, the file's buffer will be partially full, with this information not having been written to the file. If a program calls *fclose*, this function will write the partially filled buffer to the file and return an error code if this couldn't be done. If the program lets the system automatically close the file, the program won't know if an error occurred on this last write operation.

2.3 Sequential I/O

Files can be accessed sequentially and randomly. For sequential access, simply issue repeated read or write calls; each call transfers data beginning at the "current position" of the file, and updates the current position to the byte following the last byte transferred. When a file is opened, its current position is set to zero, if opened for read or write access, and to its end if opened for append.

On systems which don't keep track of the last character written to a file, such as CP/M and Apple // DOS, not all files can be correctly positioned for appending data. See the section entitled I/O for details.

2.4 Random I/O

The function *fseek* allows a file to be accessed randomly, by changing its current position. Positioning can be relative to the beginning, current position, or end of the file.

For systems which don't keep track of the last character written to a file, such as CP/M and Apple // DOS, positioning relative to the end of a file cannot always be correctly done. See the I/O overview section for details.

2.5 Buffering

When the standard i/o functions are used to access a file, the i/o is buffered. Either a user-specified or dynamically- allocated buffer can be used.

The user's program specifies a buffer to be used for a file by calling the function *setbuf* after the file has been opened but before the first i/o request to it has been made.

If, when the first i/o request is made to a file, the user hasn't specified the buffer to be used for the file, the system will automatically allocate, by calling *malloc*, a buffer for it. When the file is closed it's buffer will be freed, by calling *free*.

Dynamically allocated buffers are obtained from the one region of memory (the heap), whether requested by the standard i/o functions or by the user's program. For more information, see the overview

section *Dynamic Buffer Allocation.*

The size of an i/o buffer differs from system to system. See this chapter's system-dependent appendix for the size of this buffer on your system.

A program which both accesses files using standard i/o functions and has overlays has to take special steps to insure that an overlay won't be loaded over a buffer dynamically allocated for file i/o. For more information, see the section on overlay support in the *Technical Information* chapter.

By default, output to the console using standard i/o functions is unbuffered; all other device i/o using the standard i/o functions is buffered. Console input buffering can be disabled using the *ioctl* function; see the overview section *Console I/O* for details.

2.6 Errors

There are three fields which may be set when an exceptional condition occurs during stream i/o. Two of the fields are unique to each stream (that is, each stream has its own pair). The other is a global integer.

One of the fields associated with a stream is set if end of file is detected on input from the stream; the other is set if an error occurs during i/o to the stream. Once set for a stream, these flags remain set until the stream is closed or the program calls the *clearerr* function for the stream. The only exception to the last statement is that when called, *fseek* will reset the end of file flag for a stream. A program can check the status of the eof and error flags for a stream by calling the functions *feof* and *ferror*, respectively.

The other field which may be set is the global integer *errno*. By convention, a system function which returns an error status as its value can also set a code in *errno* which more fully defines the error. The overview section *Errors* defines the values which may be set in *errno*.

If an error occurs when a stream is being accessed, a standard i/o function returns EOF (-1) as its value, after setting a code in *errno* and setting the stream's error flag.

If end of file is reached on an input stream, a standard i/o function returns EOF after setting the stream's eof flag.

There are two techniques a program can use for detecting errors during stream i/o. First, the program can check the result of each i/o call. Second, the program can issue i/o calls and only periodically check for errors (for example, check only after all i/o is completed).

On input, a program will generally check the result of each operation.

On output to a file, a program can use either error checking technique; however, periodic checking by calling *ferror* is more efficient. When characters are written to a file using the standard i/o functions they are placed in a buffer, which is not written to disk until it is full. If the buffer isn't full, the function will return good status. It will only return bad status if the buffer was full and an error occurred while writing it to disk. Since the buffer size is 1024 bytes, most write calls will return good status, and hence periodic checking for errors is sufficient and most efficient.

Once a file opened for standard i/o is closed, *ferror* can't be used to determine if an error has occurred while writing to it. Hence *ferror* should be called after all writing to the file is completed but before the file is closed. The file should be explicitly closed by *fclose*, and its return value checked, rather than letting the system automatically close it, to know positively whether an error has occurred while writing to the file. The reason for this is that when the writing to the file is completed, it's standard i/o buffer will probably be partly full. This buffer will be written to the file when the file is closed, and *fclose* will return an error status if this final write operation fails.

2.7 The standard i/o functions

The standard i/o functions can be grouped into two sets: those that can access only the logical devices standard input, standard output, and standard error; and all the rest.

Here are the standard i/o functions that can only access *stdin*, *stdout*, and *stderr*. These are all ASCII functions; that is, they expect to deal with text characters only.

getchar	Get an ASCII character from stdin
gets	Get a line of ASCII characters from stdin
printf	Format data and send it to stdout
puterr	Send a character to stderr
putchar	Send a character to stdout
puts	Send a character string to stdout
scanf	Get a line from stdin and convert it

Here are the rest of the standard i/o functions:

agetc	Get an ASCII character
aputc	Send an ASCII character
fopen	Open a file or device
fdopen	Open as a stream a file or device already open for unbuffered i/o
freopen	Open an open stream to another file or device
fclose	Close an open stream
feof	Check for end of file on a stream
ferror	Check for error on a stream
fileno	Get file descriptor associated with stream
fflush	Write stream's buffer
fgets	Get a line of ASCII characters
fprintf	Format data and write it to a stream
fputs	Send a string of ASCII characters to a stream
fread	Read binary data
fscanf	Get data and convert it
fseek	Set current position within a file
ftell	Get current position
fwrite	Write binary data
getc	Get a binary character
getw	Get two binary characters
putc	Send a binary character
putw	Send two binary characters
setbuf	Specify buffer for stream
ungetc	Push character back into stream

3. Overview of Unbuffered I/O

The unbuffered I/O functions are used to access files and devices. They are compatible with their UNIX counterparts and are also described in chapter 8 of *The C Programming Language.*

As their name implies, a program using these functions, with two exceptions, communicates directly with files and devices; data doesn't pass through system buffers. Some unbuffered I/O, however, is buffered: when data is transferred to or from a file in blocks smaller than a certain value, it is buffered temporarily. This value differs from system to system, but is always less than or equal to 512 bytes. Also, console input can be buffered, and is, unless specific actions are taken by the user's program.

Programs which use the unbuffered i/o functions to access files generally handle the blocking and deblocking of file data themselves. Programs requiring file access but unwilling to perform the blocking and deblocking can use the standard i/o functions; see the overview section *Standard I/O* for more information.

Here are the unbuffered i/o functions:

open	Prepares a file or device for unbuffered i/o
creat	Creates a file and opens it
close	Concludes the i/o on an open file or device
read	Read data from an open file or device
write	Write data to an open file or device
lseek	Change the current position of an open file
rename	Renames a file
unlink	Deletes a file
ioctl	Change console i/o mode
isatty	Is an open file or device the console?

Before a program can access a file or device, it must be "opened", and when processing on it is done, it must be "closed".

An open file or device has an integer known as a "file descriptor" associated with it; this identifies the file or device when it's accessed.

There are two ways for a file or device to be opened for unbuffered i/o. First, it can explicitly open it, by calling the function *open.* In this case, *open* returns the file descriptor to be used when accessing the file or device.

Alternatively, the file or device can be automatically opened as one of the logical devices standard input, standard output, or standard error. In this case, the file descriptor is the integer value 0, 1, or 2, respectively. See the section entitled I/O for more information on this.

An open file or device is closed by calling the function *close.* When a program ends, any devices or files still opened for unbuffered i/o will be closed.

If an error occurs during an unbuffered i/o operation, the function returns -1 as its value and sets a code in the global integer *errno*. For more information on error handling, see the section ERRORS.

The remainder of this section discusses unbuffered i/o to files and devices.

3.1 File I/O

Programs call the functions *read* and *write* to access a file; the transfer begins at the "current position" of the file and proceeds until the number of characters specified by the program have been transferred.

The current position of a file can be manipulated in various ways by a program, allowing both sequential and random acccess to the file. For sequential access, a program simply issues consecutive i/o requests. After each operation, the current position of the file is set to the character following the last one accessed.

The function *lseek* provides random access to a file by setting the current position to a specified character location.

lseek allows the current position of a file to be set relative to the end of a file. For systems which don't keep track of the last character written to a file, such positioning cannot always be correctly done. For more information, see the section entitled I/O.

open provides a mode, O_APPEND, which causes the file being opened to be positioned at its end. This mode is supported on UNIX Systems 3 and 5, but not UNIX version 7. As with *lseek*, the positioning may not be correct for systems which don't keep track of the last character written to a file.

3.2 Device I/O

3.2.1 Unbuffered I/O to the Console

There are several options available when accessing the console, which are discussed in detail in the Console I/O sections of this chapter and of the system-dependent appendix to this chapter. Here we just want to briefly discuss the line- or character-modes of console I/O as they relate to the unbuffered i/o functions.

Console input can be either line- or character-oriented. With line-oriented input, characters are read from the console into an internal buffer a line at a time, and returned to the program from this buffer. Line buffering of console input is available even when using the so-called "unbuffered" i/o functions.

With character-oriented input, characters are read and returned to the program when they are typed: no buffering of console input occurs.

3.2.2 Unbuffered I/O to Non-Console Devices

Unbuffered I/O to devices other than the console is truly unbuffered.

4. Overview of Console I/O

A program has control over several options relating to console i/o. The primary option allows console input to be either line- or character-oriented, as described below.

On most systems, a program can selectively enable and disable the echoing of typed characters to the screen; this is called the ECHO option. A program can also enable and disable the conversion of carriage return to newline on input and of newline to carriage return-linefeed on output; this is called the CRMOD option.

On some systems, additional options are available. If your system supports additional options, they are discussed in the system dependent appendix to this chapter.

All the console i/o options have default settings, which allow a program to easily access the console without having to set the options itself. In the default mode, console i/o is line-oriented, with ECHO and CRMOD enabled.

A program can easily change the console i/o options, by calling the function *ioctl*.

Console i/o behaves the same on all systems when the console options have their default settings. However, the behavior of console i/o differs from system to system when the options are changed from their default values. Thus, a program requiring machine independence should either use the console in its default mode or be careful how it sets the console options. In the paragraphs below, we will try to point out system dependencies.

4.1 Line-oriented input

With line-oriented input, a program issuing a read request to the console will wait until an entire line has been typed. On some systems a non-UNIX option (NODELAY) is available that will prevent this waiting. If this option is available on your system, it's discussed in the system-dependent appendix to this chapter.

The program need not read an entire line at once; the line will be internally buffered, and characters returned to the program from the buffer, as requested. When the program issues a read request to the console and the buffer is empty, the program will wait until an entire new line has been typed and stored in the internal buffer (again, on some systems programs can disable this wait by setting the non-UNIX NODELAY option).

A single unbuffered read operation can return at most one line.

On most systems , selecting line-oriented console input forces the ECHO option to be enabled. On such systems the program still has control over the CRMOD option. To find out if, on your system,

line-oriented mode always has ECHO enabled, see the system-dependent appendix to this chapter.

4.2 Character-oriented input

The basic idea of character-oriented console input is that a program can read characters from the console without having to wait for an entire line to be entered.

The behavior of character-oriented console input differs from system to system, so programs requiring both machine independence and character-oriented console input have to be careful in their use of the console. However, it is possible to write such programs, although they may not be able to take full advantage of the console i/o features available for a particular system.

There are two varieties of character-oriented console input, named CBREAK and RAW. Their primary difference is that with the console in CBREAK mode, a program still has control over the other console options, whereas with the console in RAW mode it doesn't. In RAW mode, all other console options are reset: ECHO and CRMOD are disabled.

Thus, to some extent RAW mode is simply an abbreviation for 'CBREAK on, all other options off'. However, there are some differences on some systems, as noted below and in this chapter's system-dependent appendix.

The system-dependent appendix to this chapter, which accompanies your manual, presents information about character-oriented console that is specific to your system.

4.2.1 Writing system-independent programs

To write system-independent programs that access the console in character-oriented input mode, the console should be set in RAW mode, and the program should read only a single character at a time from the console. All the non-UNIX options that are supported by some systems should be reset.

The standard i/o functions all read just one character at a time from the console, even when the calling program requests several characters. Thus, programs requiring system independence and character-oriented input can read the console using the standard i/o functions.

Some systems require a program that wants to set console option to first call *ioctl* to fetch the current console options, then modify them as desired, and finally call *ioctl* to reset the new console options. The systems that don't require this don't care if a program first fetches the console options and then modifies them. Thus, a program requiring system-independence and console i/o options other than the default should fetch the current console options before modifying them.

4.3 Using ioctl

A program selects console I/O modes using the function *ioctl*. This has the form:

```
#include <sgtty.h>

ioctl(fd, code, arg)
struct sgttyb *arg;
```

The header file *sgtty.h* defines symbolic values for the *code* parameter (which tells *ioctl* what to do) and the structure *sgttyb*.

The parameter *fd* is a file descriptor associated with the console. On UNIX, this parameter defines the file descriptor associated with the device to which the *ioctl* call applies. Here, *ioctl* always applies to the console.

The parameter *code* defines the action to be performed by *ioctl*. It can have these values:

TIOCGETP	Fetch the console parameters and store them in the structure pointed at by *arg*.
TIOCSETP	Set the console parameters according to the structure pointed at by *arg*.
TIOCSETN	Equivalent to *TIOCSETP*.

The argument *arg* points to a structure named *sgttyb* that contains the following fields:

```
int sg_flags;
char sg_erase;
char sg_kill;
```

The order of these fields is system-dependent.

The *sg_flags* field is supported by all systems, while the other fields are not supported by some systems. If these fields are supported on your system, the system-dependent appendix to this chapter that accompanies your manual says so, and describes them.

To set console options, a program should fetch the current state of the sgtty fields, using ioctl's TIOCGETP option. Then it should modify the fields to the appropriate values and call ioctl again, using ioctl's TIOCSETP option.

4.4 The *sgtty* fields

4.4.1 The sg_flags field

sg_flags contains the following UNIX-compatible flags:

RAW	Set RAW mode (turns off other options). By default, RAW is disabled.
CBREAK	Return each character as soon as typed. By default, CBREAK is disabled.

ECHO	Echo input characters to the display. By default, ECHO is enabled.
CRMOD	Map CR to LF on input; convert LF to CR-LF on output. By default, CRMOD is enabled.

On some systems, other flags are contained in *sg_flags*. If your system supports other flags, they're described in the system-dependent appendix to this chapter that accompanies your manual.

More than one flag can be specified in a single call to *ioctl*; the values are simply 'or'ed together. If the RAW option is selected, none of the other options have any effect.

When the console i/o options are set and RAW and CBREAK are reset, the console is set in line-oriented input mode.

4.5 Examples

4.5.1 Console input using default mode

The following program copies characters from stdin to stdout. The console is in default mode, and assuming these streams haven't been redirected by the operator, the program will read from the keyboard and write to the display. In this mode, the operator can use the operating system's line editing facilities, such as backspace, and characters entered on the keyboard will be echoed to the display. The characters entered won't be returned to the program until the operator depresses carriage return.

```
#include <stdio.h>

main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

4.5.2 Console input - RAW mode

In this example, a program opens the console for standard i/o, sets the console in RAW mode, and goes into a loop, waiting for characters to be read from the console and then processing them. The characters typed by the operator aren't displayed unless the program itself displays them. The input request won't terminate until a character is received. This example assumes that the console is named 'con:'; on systems for which this is not the case, just substitute the appropriate name.

```
#include <stdio.h>
#include <sgtty.h>
main()
{
        int c;
        FILE *fp;
        struct sgttyb stty;

        if ((fp = fopen("con:", "r") == NULL){
                printf("can't open the console\n");
                exit();
        }

        ioctl(fileno(fp),TIOCGETP, &stty);

        stty.sg_flags |= RAW;
        ioctl(fileno(fp), TIOCSETP, &stty);
        for (;;){
                c = getc(fp);
                . . .
        }
}
```

4.5.3 Console input - console in CBREAK + ECHO mode

This example modifies the previous program so that characters read from the console are automatically echoed to the display. The program accesses the console via the standard input device. It uses the function *isatty* to verify that stdin is associated with the console; if it isn't, the program reopens stdin to the console using the function *freopen.* Again, the console is assumed to be named *con:*.

```
#include <stdio.h>
#include <sgtty.h>
main()
{
        int c;
        struct sgttyb stty;

        if (!isatty(stdin))
                freopen("con:", "r", stdin);
        ioctl(0, TIOCGETP, &stty);
        stty.sg_flags |= CBREAK | ECHO;
        ioctl(0, TIOCSETP, &stty);
        for (;;){
                c = getchar();
                . . .
        }
}
```

5. Overview of Dynamic Buffer Allocation

Several functions are provided for the dynamic allocation and deallocation of buffers from a section of memory called the 'heap'. They are:

malloc	Allocates a buffer
calloc	Allocates a buffer and initializes it to zeroes
realloc	Allocates more space to a previously allocated buffer
free	Releases an allocated buffer for reuse

These standard UNIX functions are described in the System Independent Functions section of this chapter.

In addition, on some systems the UNIX-compatible functions *sbrk* and *brk* are provided that provide a more elementary means to allocate heap space. The *malloc*-type functions call *sbrk* to get heap space, which they then manage.

On some systems, non-UNIX memory allocation functions are also supported. If such functions are supported on your system, they are described in the system-dependent appendix to this chapter that accompanies your manual.

Dynamic allocation of standard i/o buffers

Buffers used for standard i/o are dynamically allocated from the heap unless specific actions are taken by the user's program. Standard i/o calls to dynamically allocate and deallocate buffers can be interspersed with those of the user's program.

Programs which perform standard i/o and which must have absolute control of the heap can explicitly define the buffers to be used by a standard i/o stream.

Where to go from here

For descriptions of the *sbrk* and *brk* functions and, when applicable, non-UNIX memory allocation functions see the System Dependent Functions chapter.

For a discussion of i/o buffer allocation, see the Standard I/O section of the Library Functions Overviews chapter.

For more information on the heap, see the Program Organization section of the Technical Information chapter.

6. Overview of Error Processing

This section discusses error processing which relates to the global integer *errno*. This variable is modified by the standard i/o, unbuffered i/o, and scientific (eg, *sin*, *sqrt*) functions as part of their error processing.

The handling of floating point exceptions (overflow, underflow, and division by zero) is discussed in the Tech Info chapter.

When a standard i/o, unbuffered i/o, or scientific function detects an error, it sets a code in *errno* which describes the error. If no error occurs, the scientific functions don't modify *errno*. If no error occurs, the i/o functions may or may not modify *errno*.

Also, when an error occurs,

* A standard i/o function returns -1 and sets an error flag for the stream on which the error occurred;
* An unbuffered i/o function returns -1;
* A scientific function returns an arbitrary value.

When performing scientific calculations, a program can check *errno* for errors as each function is called. Alternatively, since *errno* is modified only when an error occurs, *errno* can be checked only after a sequence of operations; if it's non-zero, then an error has occurred at some point in the sequence. This latter technique can only be used when no i/o operations occur during the sequence of scientific function calls.

Since *errno* may be modified by an i/o function even if an error didn't occur, a program can't perform a sequence of i/o operations and then check *errno* afterwards to detect an error. Programs performing unbuffered i/o must check the result of each i/o call for an error.

Programs performing standard i/o operations cannot, following a sequence of standard i/o calls, check *errno* to see if an error occurred. However, associated with each open stream is an error flag. This flag is set when an error occurs on the stream and remains set until the stream is closed or the flag is explicitly reset. Thus a program can perform a sequence of standard i/o operations on a stream and then check the stream's error flag. For more details, see the standard i/o overview section.

The following table lists the system-independent values which may be placed in *errno*. These symbolic values are defined in the file *errno.h*. Other, system-dependent, values may also be set in *errno* following an i/o operation; these are error codes returned by the operating system. System dependent error codes are described in the operating system manual for a particular system.

The system-independent error codes and their meanings are:

error code	*meaning*
ENOENT	File does not exist
E2BIG	Not used
EBADF	Bad file descriptor - file is not open or improper operation requested
ENOMEM	Insufficient memory for requested operation
EEXIST	File already exists on creat request
EINVAL	Invalid argument
ENFILE	Exceeded maximum number of open files
EMFILE	Exceeded maximum number of file descriptors
ENOTTY	Ioctl attempted on non-console
EACCES	Invalid access request
ERANGE	Math function value can't be computed
EDOM	Invalid argument to math function

SYSTEM-INDEPENDENT FUNCTIONS

Chapter Contents

System Independent Functions ..... lib
Index 5
The functions 8

System Independent Functions

This chapter describes in detail the functions which are UNIX-compatible and which are common to all Aztec C packages.

The chapter is divided into sections, each of which describes a group of related functions. Each section has a name, and the sections are ordered alphabetically by name. Following this introduction is a cross reference which lists each function and the name of the section in which it is described.

A section is organized into the following subsections:

TITLE

Lists the name of the section, a phrase which is intended to catagorize the functions described in the section, and one or more letters in parentheses which specify the libraries containing the section's functions.

The letters which may appear in parentheses and their corresponding libraries are:

C	c.lib
M	m.lib

On some systems, the actual library name may be a variant on the name given above. For example, on TRSDOS, the libraries are named *c/lib* and *m/lib.*

With *Apprentice C*, the functions are all in the run-time system, and not libraries.

SYNOPSIS

Indicates the types of arguments that the functions described in the section require, and the values they return. For example, the function *atof* converts character strings into double precision numbers. It is listed in the synopsis as

```
double atof(s)
char *s;
```

This means that *atof()* returns a value of type *double* and requires as an argument a pointer to a character string. Since *atof* returns a non-integer value, prior to use of the function it must be declared:

```
double atof();
```

The notation

```
#include "header.h"
```

at the beginning of a synopsis indicates that such a statement should appear at the beginning of any program calling one of the functions described in the section.

On Radio Shack systems, a header file can use either a period or a slash to separate the filename from the extent. That is, the include statement can be as listed above, or

```
#include "header/h"
```

DESCRIPTION

Describes the section's functions.

SEE ALSO

Lists relevant sections. A letter in parentheses may follow a section name. This specifies where the section is located: no letter means that the section is in the current chapter; 'O' means that it's in the Functions Overview chapter; 'S' means that it's in the System Dependent Functions chapter.

DIAGNOSTICS

Describes the error codes that the section's functions may return. The section ERRORS in the Functions Overview chapter presents an overview of error processing.

EXAMPLES

Gives examples on use of the section's functions.

Index to System Independent Functions

function	*page*	*description*
acos	SIN	compute arccosine
agetc	GETC	get ASCII char from a stream
aputc	PUTC	put ASCII char to a stream
asin	SIN	compute arcsine
atan	SIN	compute arctangent
atan2	SIN	another arctangent function
atof	ATOF	convert char string to a *double*
atoi	ATOF	convert char string to an *int*
atol	ATOF	convert char string to a *long*
calloc	MALLOC	allocate a buffer
ceil	FLOOR	get smallest integer not less than x
clearerr	FERROR	clear error flags on a stream
close	CLOSE	close of unbuffered file/device
cos	SIN	compute cosine
cosh	SINH	compute hyperbolic cosine
cotan	SIN	compute cotangent
creat	CREAT	create a file & open for unbuffered i/o
exp	EXP	compute exponential
fabs	FLOOR	compute absolute value
fclose	FCLOSE	close i/o stream
fdopen	FOPEN	open file descriptor as an i/o stream
feof	FERROR	check for eof on an i/o stream
ferror	FERROR	check for error on an i/o stream
fflush	FCLOSE	flush an i/o stream
fgets	GETS	get a line from an i/o stream
fileno	FERROR	get file descriptor for i/o stream
floor	FLOOR	get largest *int* not greater than x
fopen	FOPEN	open i/o stream
format	PRINTF	formatting utility for *printf*
fprintf	PRINTF	format string & send to i/o stream
fputs	PUTS	put char string to i/o stream
fread	FREAD	read binary data from i/o stream
free	MALLOC	release buffer
freopen	FOPEN	reopen i/o stream
frexp	FREXP	get components of a *double*
fscanf	SCANF	input string from i/o stream & convert
fseek	FSEEK	position i/o stream
ftell	FSEEK	determine position in i/o stream
ftoa	ATOF	convert float/double to char string

fwrite	FREAD	write binary data to i/o stream
getc	GETC	get binary char from i/o stream
getchar	GETC	get ASCII char from stdin
gets	GETS	get ASCII line from stdin
getw	GETW	get ASCII word from stdin
index	STRING	find char in string
ioctl	IOCTL	set mode of device
isalpha, etc.	CTYPE	char classification functions
isatty	IOCTL	is this a console?
ldexp	FREXP	build *double*
log	EXP	compute natural logarithm
log10	EXP	compute base-10 log
longjmp	SETJMP	non-local goto
lseek	LSEEK	position unbuffered i/o file
malloc	MALLOC	allocate buffer
movmem	MOVMEM	copy a block of memory
modf	FREXP	get components of *double*
open	OPEN	open file/device for unbuffered i/o
pow	EXP	compute x**y
printf	PRINTF	format data and print on stdout
putc	PUTC	put binary char to i/o stream
putchar	PUTC	put ASCII char to stdout
puterr	PUTC	put ASCII char to stderr
puts	PUTS	put ASCII string to stdout
putw	PUTC	put ASCII word to stdout
qsort	QSORT	Quick sort
ran	RAN	compute random number
read	READ	read unbuffered file/device
realloc	MALLOC	reallocate buffer
rename	RENAME	rename file
rindex	STRING	find char in string
scanf	SCANF	input string from stdin & convert
setbuf	SETBUF	set buffer for i/o stream
setjmp	SETJMP	*longjmp* partner
setmem	MOVMEM	set memory to specified byte
sin	SIN	compute sine
sinh	SINH	compute hyperbolic sine
sprintf	PRINTF	format string into buffer
sqrt	EXP	compute square root
sscanf	SCANF	convert string from buffer
strcat	STRING	concatenate two strings
strcmp	STRING	compare two strings
strcpy	STRING	copy char string
strlen	STRING	get length of char string
strncat	STRING	concatenate strings
strncmp	STRING	compare strings
strncpy	STRING	copy string
swapmem	MOVMEM	swap two blocks of memory

tan	SIN	compute tangent
tanh	SINH	compute hyperbolic tangent
tolower	TOUPPER	convert upper case char to lower
toupper	TOUPPER	convert lower case char to upper
ungetc	UNGETC	return char to i/o stream
unlink	UNLINK	delete file
write	WRITE	unbuffered write of binary data

NAME

atof, atoi, atol - convert ASCII to numbers
ftoa - convert floating point to ASCII

SYNOPSIS

```
double atof(cp)
char *cp;

atoi(cp)
char *cp;

long atol(cp)
char *cp;

ftoa(val, buf, precision, type)
double val;
char *buf;
int precision, type;
```

DESCRIPTION

atof, *atoi*, and *atol* convert a string of text characters pointed at by the argument *cp* to double, integer, and long representations, respectively.

atof recognizes a string containing leading blanks and tabs, which it skips, then an optional sign, then a string of digits optionally containing a decimal point, then an optional 'e' or 'E' followed by an optionally signed integer.

atoi and *atol* recognize a string containing leading blanks and tabs, which are ignored, then an optional sign, then a string of digits.

ftoa converts a double precision floating point number to ASCII. *val* is the number to be converted and *buf* points to the buffer where the ASCII string will be placed. *precision* specifies the number of digits to the right of the decimal point. *type* specifies the format: 0 for "E" format, 1 for "F" format, 2 for "G" format.

atof and *ftoa* are in the library *m.lib*; the other functions are in *c.lib*.

NAME

close - close a device or file

SYNOPSIS

close(fd)
int fd;

DESCRIPTION

close closes a device or disk file which is opened for unbuffered i/o.

The parameter *fd* is the file descriptor associated with the file or device. If the device or file was explicitly opened by the program by calling *open* or *creat*, *fd* is the file descriptor returned by *open* or *creat*.

close returns 0 as its value if successful.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *close* fails, it returns -1 and sets an error code in the global integer *errno*.

NAME

creat - create a new file

SYNOPSIS

```
creat(name, pmode)
char *name;
int pmode;
```

DESCRIPTION

creat creates a file and opens it for unbuffered, write-only access. If the file already exists, it is truncated so that nothing is in it (this is done by erasing and then creating the file).

creat returns as its value an integer called a "file descriptor". Whenever a call is made to one of the unbuffered i/o functions to access the file, its file descriptor must be included in the function's parameters.

name is a pointer to a character string which is the name of the device or file to be opened. See the I/O overview section for details.

For most systems, *pmode* is optional: if specified, it's ignored. It should be included, however, for programs for which UNIX-compatibility is required, since the UNIX creat function requires it. In this case, *pmode* should have the octal value 0666.

For some systems, *pmode* is required and has a special meaning. If it is required for your system, the System Dependent Functions chapter will contain a description of the *creat* function, which will define the meaning.

SEE ALSO

Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If *creat* fails, it returns -1 as its value and sets a code in the global integer *errno*.

NAME

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii
- character classification functions

SYNOPSIS

#include "ctype.h"

isalpha(c)

. . .

DESCRIPTION

These macros classify ASCII-coded integer values by table lookup, returning nonzero if the integer is in the catagory, zero otherwise. *isascii* is defined for all integer values. The others are defined only when *isascii* is true and on the single non-ASCII value EOF (-1).

isalpha	*c* is a letter
isupper	*c* is an upper case letter
islower	*c* is a lower case letter
isdigit	*c* is a digit
isalnum	*c* is an alphanumeric character
isspace	*c* is a space, tab, carriage return, newline, or formfeed
ispunct	*c* is a punctuation character
isprint	*c* is a printing character, valued 0x20 (space) through 0x7e (tilde)
iscntrl	*c* is a delete character (0xff) or ordinary control character (value less than 0x20)
isascii	*c* is an ASCII character, code less than 0x100

NAME

exponential, logarithm, power, square root functions:
exp, log, log10, pow, sqrt

SYNOPSIS

```
#include <math.h>

double exp(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double pow(x, y)
double x,y;

double sqrt(x)
double x;
```

DESCRIPTION

exp returns the exponential function of x.

log returns the natural logarithm of x; *log10* returns the base 10 logarithm.

pow returns x ** y (x to the y-th power).

sqrt returns the square root of x.

SEE ALSO

Errors (O)

DIAGNOSTICS

If a function can't perform the computation, it sets an error code in the global integer *errno* and returns an arbitrary value; otherwise it returns the computed value without modifying *errno*. The symbolic values which a function can place in *errno* are EDOM, signifying that the argument was invalid, and ERANGE, meaning that the value of the function couldn't be computed. These codes are defined in the file *errno.h*.

The following table lists, for each function, the error codes that can be returned, the function value for that error, and the meaning of the error. The symbolic values are defined in the file *math.h*.

function	error	f(x)	Meaning
exp	ERANGE	HUGE	x > LOGHUGE
"	ERANGE	0.0	x < LOGTINY
log	EDOM	-HUGE	x <= 0
log10	EDOM	-HUGE	x <= 0
pow	EDOM	-HUGE	x < 0, x=y=0
"	ERANGE	HUGE	y*log(x)>LOGHUGE
"	ERANGE	0.0	y*log(x)<LOGTINY
sqrt	EDOM	0.0	x < 0.0

NAME

fclose, fflush - close or flush a stream

SYNOPSIS

```
#include "stdio.h"

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

fclose informs the system that the user's program has completed its buffered i/o operations on a device or file which it had previously opened (by calling *fopen*). *fclose* releases the control blocks and buffers which it had allocated to the device or file. Also, when a file is being closed, *fclose* writes any internally buffered information to the file.

fclose is called automatically by *exit.*

fflush causes any buffered information for the named output stream to be written to that file. The stream remains open.

If *fclose* or *fflush* is successful, it returns 0 as its value.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

If the operation fails, -1 is returned, and an error code is set in the global integer *errno.*

NAME

feof, ferror, clearerr, fileno - stream status inquiries

SYNOPSIS

```
#include "stdio.h"

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream;

clearerr(stream)
FILE *stream;

fileno(stream)
FILE *stream;
```

DESCRIPTION

feof returns non-zero when end-of-file is reached on the specified input stream, and zero otherwise.

ferror returns non-zero when an error has occurred on the specified stream, and zero otherwise. Unless cleared by *clearerr*, the error indication remains set until the stream is closed.

clearerr resets an error indication on the specified stream.

fileno returns the integer file descriptor associated with the stream.

These functions are defined as macros in the file stdio.h.

SEE ALSO

Standard I/O (O)

NAME

fabs, floor, ceil - absolute value, floor, ceiling routines

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

fabs returns the absolute value of *x*.

floor returns the largest integer not greater than *x*.

ceil returns the smallest integer not less than *x*.

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include "stdio.h"

FILE *fopen(filename, mode)
char *filename, *mode;

FILE *freopen(filename, mode, stream)
char *filename, *mode;
FILE *stream;

FILE *fdopen(fd, mode)
char *mode;
```

DESCRIPTION

These functions prepare a device or disk file for access by the standard i/o functions; this is called "opening" the device or file. A file or device which has been opened by one of these functions is called a "stream".

If the device or file is successfully opened, these functions return a pointer, called a "file pointer" to a structure of type FILE. This pointer is included in the list of parameters to buffered i/o functions, such as *getc* or *putc*, which the user's program calls to access the stream.

fopen is the most basic of these functions: it simply opens the device or file specified by the *filename* parameter for access specified by the *mode* parameter. These parameters are described below.

freopen substitutes the named device or file for the device or file which was previously associated with the specified stream. It closes the device or file which was originally associated with the stream and returns *stream* as its value. It is typically used to associate devices and files with the preopened streams *stdin*, *stdout*, and *stderr*.

fdopen opens a device or file for buffered i/o which has been previously opened by one of the unbuffered open functions *open* and *creat*. It returns as it's value a FILE pointer.

fdopen is passed the file descriptor which was returned when the device or file was opened by *open* or *creat*. It's also passed the *mode* parameter specifying the type of access desired. *mode* must agree with the mode of the open file.

The parameter *filename* is a pointer to a character string which is the name of the device or file to be opened. For details, see the I/O overview section.

mode points to a character string which specifies how the user's program intends to access the stream. The choices are as follows:

mode	*meaning*
r	Open for reading only. If a file is opened, it is positioned at the first character in it. If the file or device does not exist, NULL is returned.
w	Open for writing only. If a file is opened which already exists, it is truncated to zero length. If the file does not exist, it is created.
a	Open for appending. The calling program is granted write-only access to the stream. The current file position is the character after the last character in the file. If the file does not exist, it is created.
x	Open for writing. The file must not previously exist. This option is not supported by Unix.
r+	Open for reading and writing. Same as "r", but the stream may also be written to.
w+	Open for writing and reading. Same as "w", but the stream may also be read; different from "r+" in the creation of a new file and loss of any previous one.
a+	Open for appending and reading. Same as "a", but the stream may also be read; different from "r+" in file positioning and file creation.
x+	Open for writing and reading. Same as "x" but the file can also be read.

On systems which don't keep track of the last character in a file (for example CP/M and Apple DOS), not all files can be correctly positioned when opened in append mode. See the I/O overview section for details.

SEE ALSO

I/O (O), Standard I/O (O)

DIAGNOSTICS

If the file or device cannot be opened, NULL is returned and an error code is set in the global integer *errno*.

EXAMPLES

The following example demonstrates how *fopen* can be used in a program.

```
#include "stdio.h"

main(argc,argv)
char **argv;
{
    FILE *fopen(), *fp;

    if ((fp = fopen(argv[1], argv[2])) == NULL)  {
        printf("You asked me to open %s",argv[1]);
        printf("in the %s mode", argv[2]);
        printf("but I can't!\n");
    } else
        printf("%s is open\n", argv[1]);
```

Here is a program which uses *freopen*:

```
#include "stdio.h"
main()
{
    FILE *fp;
    fp = freopen("dskfile", "w+", stdout);
    printf("This message is going to dskfile\n");
}
```

Here is a program which uses *fdopen*:

```
#include "stdio.h"

dopen_it(fd)
int fd;  /* value returned by previous call to open */
{
    FILE *fp;

    if ((fp = fdopen(fd, "r+")) == NULL)
        printf("can't open file for r+\n");
    else
        return(fp);
}
```

NAME

fread, fwrite - buffered binary input/output

SYNOPSIS

```
#include "stdio.h"

int fread(buffer, size, count, stream)
char *buffer;
int size, count;
FILE *stream;

int fwrite(buffer, size, count, stream)
char *buffer;
int size, count;
FILE *stream;
```

DESCRIPTION

fread performs a buffered input operation and *fwrite* a buffered write operation to the open stream specified by the parameter *stream*.

buffer is the address of the user's buffer which will be used for the operation.

The function reads or writes *count* items, each containing *size* bytes, from or to the stream.

fread and *fwrite* perform i/o using the functions *getc* and *putc*; thus, no translations occur on the data being transferred.

The function returns as its value the number of items actually read or written.

SEE ALSO

Standard I/O (O), Errors (O), fopen, ferror

DIAGNOSTICS

fread and *fwrite* return 0 upon end of file or error. The functions *feof* and *ferror* can be used to distinguish between the two. In case of an error, the global integer *errno* contains a code defining the error.

EXAMPLE

This is the code for reading ten integers from file 1 and writing them again to file 2. It includes a simple check that there are enough two-byte items in the first file:

```
#include "stdio.h"

main()
{
    FILE *fp1, *fp2, *fopen();
    char *buf;
    int size = 2, count = 10;

    fp1 = fopen("file1","r");
    fp2 = fopen("file2","w");
    if (fread(buf, size, count, fp1) != count)
        printf("Not enough integers in file1\n");
    fwrite(buf, size, count, fp2);
}
```

NAME

frexp, ldexp, modf - build and unbuild real numbers

SYNOPSIS

```
#include <math.h>

double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

Given *value*, *frexp* computes integers x and n such that value=x*2**n. x is returned as the value of *frexp*, and n is stored in the *int* field pointed at by *eptr*.

ldexp returns the double quantity value*2**exp.

modf returns as its value the positive fractional part of value and stores the integer part in the double field pointed at by *iptr*.

NAME

fseek, ftell - reposition a stream

SYNOPSIS

```
#include "stdio.h"

int fseek(stream, offset, origin)
FILE *stream;
long offset;
int origin;

long ftell(stream)
FILE *stream;
```

DESCRIPTION

fseek sets the "current position" of a file which has been opened for buffered i/o. The current position is the byte location at which the next input or output operation will begin.

stream is the stream identifier associated with the file, and was returned by *fopen* when the file was opened.

offset and *origin* together specify the current position: the new position is at the signed distance *offset* bytes from the beginning, current position, or end of the file, depending on whether *origin* is 0, 1, or 2, respectively.

offset can be positive or negative, to position after or before the specified origin, respectively, with the limitation that you can't seek before the beginning of the file.

For some operating systems (for example, CP/M and Apple DOS) a file may not be able to be correctly positioned relative to its end. See the overview sections I/O and STANDARD I/O for details.

If *fseek* is successful, it will return zero.

ftell returns the number of bytes from the beginning to the current position of the file associated with *stream.*

SEE ALSO

Standard I/O (O), I/O (O), lseek

DIAGNOSTICS

fseek will return -1 for improper seeks. In this case, an error code is set in the global integer *errno.*

EXAMPLE

The following routine is equivalent to opening a file in "a+" mode:

```
a_plus(filename)
char *filename;
{
    FILE *fp, *fopen();

    if ((fp = fopen(filename, r+)) == NULL)
    fp = fopen(filename, w+);
    fseek(fp, 0L, 2);   /* position 1 byte past
                           last character */
}
```

To set the current position back 5 characters before the present current position, the following call can be used:

```
fseek(fp, -5L, 1)
```

NAME

getc, agetc, getchar, getw

SYNOPSIS

```
#include "stdio.h"

int getc(stream)
FILE *stream;

int agetc(stream)       /* non-Unix function */
FILE *stream;

int getchar()

int getw(stream)
FILE *stream;
```

DESCRIPTION

getc returns the next character from the specified input stream.

agetc is used to access files of text. It generally behaves like *getc* and returns the next character from the named input stream. It differs from *getc* in the following ways:

* It translates end-of-line sequences (eg, carriage return on Apple DOS; carriage return-line feed on CP/M) to a single newline ('\\n') character.
* It translates an end-of-file sequence (eg, a null character on Apple DOS; a control-z character on CP/M) to EOF;
* It ignores null characters (' ') on all systems except Apple DOS;
* On some systems, the most significant bit of each character returned is set to zero.

agetc is not a UNIX function. It is, however, provided with all Aztec C packages, and provides a convenient, system-independent way for programs to read text.

getchar returns text characters from the standard input stream (stdin). It is implemented as the call *agetc(stdin).*

getw returns the next word from the specified input stream. It returns EOF (-1) upon end-of-file or error, but since that is a good integer value, *feof* and *ferror* should be used to check the success of *getw*. It assumes no special alignment in the file.

SEE ALSO

I/O (O), Standard I/O (O), fopen, fclose

DIAGNOSTICS

These functions return EOF (-1) at end of file or if an error occurs. The functions *feof* and *ferror* can be used to distinguish the two. In the latter case, an error code is set in the global

integer *errno*.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include "stdio.h"

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

DESCRIPTION

gets reads a string of characters from the standard input stream, stdin, into the buffer pointed by *s*. The input operation terminates when either a newline character (\\n) or end of file is encountered.

fgets reads characters from the specified input stream into the buffer pointer at by *s* until either (1) *n*-1 characters have been read, (2) a newline character (\\n) is read, or (3) end of file or an error is detected on the stream.

Both functions return *s*, except as noted below.

gets and *fgets* differ in their handling of the newline character: *gets* doesn't put it in the caller's buffer, while *fgets* does. This is the behavior of these functions under UNIX.

These functions get characters using *agetc*; thus they can only be used to get characters from devices and files which contain text characters.

SEE ALSO

I/O (O), Standard I/O (O), ferror

DIAGNOSTICS

gets and *fgets* return the pointer NULL (0) upon reaching end of file or detecting an error. The functions *feof* and *ferror* can be used to distinguish the two. In the latter case, an error code is placed in the global integer *errno*.

NAME

ioctl, isatty - device i/o utilities

SYNOPSIS

```
#include "sgtty.h"

ioctl(fd, cmd, stty)
struct sgttyb *stty;

isatty(fd)
```

DESCRIPTION

ioctl sets and determines the mode of the console.

For more details on *ioctl*, see the overview section on console I/O.

isatty returns non-zero if the file descriptor *fd* is associated with the console, and zero otherwise.

SEE ALSO

Console I/O (O)

NAME

lseek - change current position within file

SYNOPSIS

long int lseek(fd, offset, origin)
int fd, origin;
long offset;

DESCRIPTION

lseek sets the current position of a file which has been opened for unbuffered i/o. This position determines where the next character will be read or written.

fd is the file descriptor associated with the file.

The current position is set to the location specified by the offset and origin parameters, as follows:

* If *origin* is 0, the current position is set to *offset* bytes from the beginning of the file.
* If *origin* is 1, the current position is set to the current position plus *offset*.
* If *origin* is 2, the current position is set to the end of the file plus *offset*.

The offset can be positive or negative, to position after or before the specified origin, respectively.

Positioning of a file relative to its end (that is, calling *lseek* with *origin* set to 2) cannot always be correctly done on all systems (for example, CP/M and Apple DOS). See the section entitled I/O for details.

If *lseek* is successful, it will return the new position in the file (in bytes from the beginning of the file).

SEE ALSO

Unbuffered I/O (O), I/O (O)

DIAGNOSTICS

If *lseek* fails, it will return -1 as its value and set an error code in the global integer *errno*. *errno* is set to EBADF if the file descriptor is invalid. It will be set to EINVAL if the offset parameter is invalid or if the requested position is before the beginning of the file.

EXAMPLES

1. To seek to the beginning of a file:

```
lseek(fd, 0L, 0);
```

lseek will return the value zero (0) since the current position in the file is character (or byte) number zero.

2. To seek to the character following the last character in the file:

```
pos = lseek(fd, 0L, 2);
```

The variable *pos* will contain the current position of the end of file, plus one.

3. To seek backward five bytes:

```
lseek(fd, -5L, 1);
```

The third parameter, 1, sets the origin at the current position in the file. The offset is -5. The new position will be the origin plus the offset. So the effect of this call is to move backward a total of five characters.

4. To skip five characters when reading in a file:

```
read(fd, buf, count);
lseek(fd, 5L, 1);
read(fd, buf, count);
```

NAME

malloc, calloc, realloc, free - memory allocation

SYNOPSIS

```
char *malloc(size)
unsigned size;

char *calloc(nelem, elemsize)
unsigned nelem, elemsize;

char *realloc(ptr, size)
char *ptr;
unsigned size;

free(ptr)
char *ptr;
```

DESCRIPTION

These functions are used to allocate memory from the "heap", that is, the section of memory available for dynamic storage allocation.

malloc allocates a block of *size* bytes, and returns a pointer to it.

calloc allocates a single block of memory which can contain *nelem* elements, each *elemsize* bytes big, and returns a pointer to the beginning of the block. Thus, the allocated block will contain (*nelem* * *elemsize*) bytes. The block is initialized to zeroes.

realloc changes the size of the block pointed at by *ptr* to *size* bytes, returning a pointer to the block. If necessary, a new block will be allocated of the requested size, and the data from the original block moved into it. The block passed to *realloc* can have been freed, provided that no intervening calls to *calloc*, *malloc*, or *realloc* have been made.

free deallocates a block of memory which was previously allocated by *malloc*, *calloc*, or *realloc*; this space is then available for reallocation. The argument *ptr* to *free* is a pointer to the block.

malloc and *free* maintain a circular list of free blocks. When called, *malloc* searches this list beginning with the last block freed or allocated coalescing adjacent free blocks as it searches. It allocates a buffer from the first large enough free block that it encounters. If this search fails, it calls *sbrk* to get more memory for use by these functions.

SEE ALSO

Memory Usage (O), break (S)

DIAGNOSTICS

malloc, *calloc* and *realloc* return a null pointer (0) if there is no available block of memory.

free returns -1 if it's passed an invalid pointer.

NAME

movmem, setmem, swapmem

SYNOPSIS

```
movmem(src, dest, length)          /* non-Unix function */
char *src, *dest;
int length;

setmem(area,length,value)          /* non-Unix function */
char *area;

swapmem(s1, s2, len)               /* non-Unix function */
char *s1, *s2;
```

DESCRIPTION

movmem copies *length* characters from the block of memory pointed at by *src* to that pointed at by *dest.*

movmem copies in such a way that the resulting block of characters at *dest* equals the original block at *src.*

setmem sets the character *value* in each byte of the block of memory which begins at *area* and continues for *length* bytes.

swapmem swaps the blocks of memory pointed at by *s1* and *s2.* The blocks are *len* bytes long.

NAME

open

SYNOPSIS

```
#include "fcntl.h"

open(name, mode)    /* calling sequence on most systems */
char *name;

/* calling sequence on some systems (see below): */
open(name, mode, param3)
char *name;
```

DESCRIPTION

open opens a device or file for unbuffered i/o. It returns an integer value called a file descriptor which is used to identify the file or device in subsequent calls to unbuffered i/o functions.

name is a pointer to a character string which is the name of the device or file to be opened. For details, see the overview section I/O.

mode specifies how the user's program intends to access the file. The choices are as follows:

mode	*meaning*
O_RDONLY	read only
O_WRONLY	write only
O_RDWR	read and write
O_CREAT	Create file, then open it
O_TRUNC	Truncate file, then open it
O_EXCL	Cause open to fail if file already exists; used with O_CREAT
O_APPEND	Position file for appending data

These open modes are integer constants defined in the files *fcntl.h*. Although the true values of these constants can be used in a given call to open, use of the symbolic names ensures compatibility with UNIX and other systems.

The calling program must specify the type of access desired by including exactly one of O_RDONLY, O_WRONLY, and O_RDWR in the mode parameter. The three remaining values are optional. They may be included by adding them to the mode parameter, as in the examples below.

By default, the open will fail if the file to be opened does not exist. To cause the file to be created when it does not already exist, specify the O_CREAT option. If O_EXCL is given in addition to O_CREAT, the open will fail if the file already exists; otherwise, the file is created.

If the O_TRUNC option is specified, the file will be truncated so that nothing is in it. The truncation is performed by simply erasing the file, if it exists, and then creating it. So it is not an error to use this option when the file does not exist.

Note that when O_TRUNC is used, O_CREAT is not needed.

If O_APPEND is specified, the current position for the file (that is, the position at which the next data transfer will begin) is set to the end of the file. For systems which don't keep track of the last character written to a file (for example, CP/M and Apple DOS), this positioning cannot always be correctly done. See the I/O overview section for details. Also, this option is not supported by UNIX.

param3 is not needed or used on many systems. If it is needed for your system, the System Dependent Library Functions chapter will contain a description of the *open* function, which will define this parameter.

If *open* does not detect an error, it returns an integer called a "file descriptor." This value is used to identify the open file during unbuffered i/o operations. The file descriptor is very different from the file pointer which is returned by *fopen* for use with buffered i/o functions.

SEE ALSO

I/O (O), Unbuffered I/O (O), Errors (O)

DIAGNOSTICS

If open encounters an error, it returns -1 and sets the global integer *errno* to a symbolic value which identifies the error.

EXAMPLES

1. To open the file, testfile, for read-only access:

```
fd = open("testfile", O_RDONLY);
```

If testfile does not exist *open* will just return -1 and set *errno* to ENOENT.

2. To open the file, sub1, for read-write access:

```
fd = open("sub1", O_RDWR+O_CREAT);
```

If the file does not exist, it will be created and then opened.

3. The following program opens a file whose name is given on the command line. The file must not already exist.

```
main(argc, argv)
char **argv;
{
    int fd;

    fd = open(*++argv, O_WRONLY+O_CREAT+O_EXCI
    if (fd = -1)  {
    if (errno == EEXIST)
    printf("file already exists\n");
    else if (errno == ENOENT)
        printf("unable to open file\n");
    else
        printf("open error\n");
}
```

NAME

printf, fprintf, sprintf, format
- formatted output conversion functions

SYNOPSIS

```
#include "stdio.h"

printf(fmt [,arg] ...)
char *fmt;

fprintf(stream, fmt [,arg] ...)
FILE *stream;
char *fmt;

sprintf(buffer, fmt [,arg] ...)
char *buffer, *fmt;

format(func, fmt, argptr)
int (*func)();
char *fmt;
unsigned *argptr;
```

DESCRIPTION

These functions convert and format their arguments (*arg* or *argptr*) according to the format specification *fmt*. They differ in what they do with the formatted result:

printf outputs the result to the standard output stream, *stdout*;

fprintf outputs the result to the stream specified in its first argument, *stream*;

sprintf places the result in the buffer pointed at by its first argument, *buffer*, and terminates the result with the null character, ' '.

format calls the function *func* with each character of the result. In fact, *printf*, *fprintf*, and *sprintf* call *format* with each character that they generate.

These functions are in both *c.lib* and *m.lib*, the difference being that the *c.lib* versions don't support floating point conversions. Hence, if floating point conversion is required, the *m.lib* versions must be used. If floating point conversion isn't required, either version can be used. To use *m.lib*'s version, *m.lib* must be specified before *c.lib* at the time the program is linked.

The character string pointed at by the *fmt* parameter, which directs the print functions, contains two types of items: ordinary characters, which are simply output, and conversion specifications, each of which causes the conversion and output of the next successive *arg*.

A conversion specification begins with the character % and continues with:

* An optional minus sign (-) which specifies left adjustment of the converted value in the output field;
* An optional digit string specifying the 'field width' for the conversion. If the converted value has fewer characters than this, enough blank characters will be output to make the total number of characters output equals the field width. If the converted value has more characters than the field width, it will be truncated. The blanks are output before or after the value, depending on the presence or absence of the left- adjustment indicator. If the field width digits have a leading 0, 0 is used as a pad character rather than blank.
* An optional period, '.', which separates the field width from the following field;
* An optional digit string specifying a precision; for floating point conversions, this specifies the number of digits to appear after the decimal point; for character string conversions, this specifies the maximum number of characters to be printed from a string;
* Optionally, the character *l*, which specifies that a conversion which normally is performed on an *int* is to be performed on a *long*. This applies to the d, o, and x conversions.
* A character which specifies the type of conversion to be performed.

A field width or precision may be * instead of a number, specifying that the next available *arg*, which must be an *int*, supplies the field width or precision.

The conversion characters are:

d, *o*, or *x*	The *int* in the corresponding *arg* is converted to decimal, octal, or hexadecimal notation, respectively, and output;
u	The unsigned integer *arg* is converted to decimal notation;
c	The character *arg* is output. Null characters are ignored;
s	The characters in the string pointed at by *arg* are output until a null character or the number of characters indicated by the precision is reached. If the precision is zero or missing, all characters in the string, up to the terminating null, are output;
f	The float or double *arg* is converted to decimal notation in the style '[-]ddd.ddd'. The number

of d's after the decimal point is equal to the precision given in the conversion specification. If the precision is missing, it defaults to six digits. If the precision is explicitly 0, the decimal point is also not printed.

e The float or double *arg* is converted to the style '[-]d.ddde[-]dd', where there is one digit before the decimal point and the number after is equal to the precision given. If the precision is missing, it defaults to six digits.

g The float or double *arg* is printed in style d, f, or e, whichever gives full precision in minimum space.

% Output a %. No argument is converted.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. The following program fragment:

```
char *name; float amt;
printf("your total, %s, is $%f\n", name, amt);
```

will print a message of the form

```
your total, Alfred, is $3.120000
```

Since the precision of the %f conversion wasn't specified, it defaulted to six digits to the right of the decimal point.

2. This example modifies example 1 so that the field width for the %s conversion is three characters, and the field width and precision of the %f conversion are 10 and 2, respectively. The %f conversion will also use 0 as a pad character, rather than blank.

```
char *name; float amt;
printf("your total, %3s, is $%10.2f\n", name, amt);
```

3. This example modifies example 2 so that the field width of the %s conversion and the precision of the %f conversion are taken from the variables nw and ap:

```
char *name; float amt; int nw, ap;
printf("your total %*s,is $%10.*f\n",nw,name,ap,amt);
```

4. This example demonstrates how to use the *format* function by listing *printf*, which calls *format* with each character that it generates.

```
printf(fmt,args)
char *fmt; unsigned args;
{
    extern int putchar();
    format(putchar,fmt,&args);
}
```

NAME

putc, aputc, putchar, putw, puterr
- put character or word to a stream

SYNOPSIS

```
#include "stdio.h"

putc(c, stream)
char c;
FILE *stream;

aputc(c, stream)            /* non-Unix function */
char c;
FILE *stream;

putchar(c)
char c;

putw(w,stream)
FILE *stream;

puterr(c)                   /* non-Unix function */
char c;
```

DESCRIPTION

putc writes the character *c* to the named output stream. It returns *c* as its value.

aputc is used to write text characters to files and devices. It generally behaves like *putc*, and writes a single character to a stream. It differs from *putc* as follows:

* When a newline character is passed to *aputc*, an end- of-line sequence (eg, carriage return followed by line feed on CP/M, and carriage return only on Apple DOS) is written to the stream;
* The most significant bit of a character is set to zero before being written to the stream.
* *aputc* is not a UNIX function. It is, however, supported on all Aztec C systems, and provides a convenient, system-independent way for a program to write text.
* *putchar* writes the character *c* to the standard output stream, stdout. It's identical to *aputc(c, stdout)*.
* *putw* writes the word *w* to the specified stream. It returns *w* as its value. *putw* neither requires nor causes special alignment in the file.
* *puterr* writes the character *c* to the standard error stream, stderr. It's identical to *aputc(c, stderr)*. It is not a UNIX function.

SEE ALSO

Standard I/O

DIAGNOSTICS

These functions return EOF (-1) upon error. In this case, an error code is set in the global integer *errno*.

NAME

puts, fputs - put a character string on a stream

SYNOPSIS

```
#include "stdio.h"

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

puts writes the null-terminated string *s* to the standard output stream, stdout, and then an end-of-line sequence. It returns a non-negative value if no errors occur.

fputs copies the null-terminated string *s* to the specified output stream. It returns 0 if no errors occur.

Both functions write to the stream using *aputc*. Thus, they can only be used to write text. See the PUTC section for more details on *aputc*.

Note that *puts* and *fputs* differ in this way: On encountering a newline character, *puts* writes an end-of-line sequence and *fputs* doesn't.

SEE ALSO

Standard I/O (O), putc

DIAGNOSTICS

If an error occurs, these functions return EOF (-1) and set an error code in the global integer *errno*.

NAME

qsort - sort an array of records in memory

SYNOPSIS

```
qsort(array, number, width, func)
char *array;
unsigned number;
unsigned width;
int (*func)();
```

DESCRIPTION

qsort sorts an array of elements using Hoare's Quicksort algorithm. *array* is a pointer to the array to be sorted; *number* is the number of record to be sorted; *width* is the size in bytes of each array element; *func* is a pointer to a function which is called for a comparison of two array elements.

func is passed pointers to the two elements being compared. It must return an integer less than, equal to, or greater than zero, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

EXAMPLE

The Aztec linker, LN, can generate a file of text containing a symbol table for a program. Each line of the file contains an address at which a symbol is located, followed by a space, followed by the symbol name. The following program reads such a symbol table from the standard input, sorts it by address, and writes it to standard output.

```
#include "stdio.h"
#define MAXLINES 2000
#define LINESIZE 16
char *lines[MAXLINES], *malloc();

main()
{
    int i,numlines, cmp();
    char buf[LINESIZE];

    for (numlines=0; numlines<MAXLINES; ++numlines){
        if (gets(buf) == NULL)
            break;
        lines[numlines] = malloc(LINESIZE);
        strcpy(lines[numlines], buf);
    }
    qsort(lines, numlines, 2, cmp);
    for (i = 0; i <numlines; ++i)
        printf("%s\n", lines[i]);
}

cmp(a,b)
char **a, **b;
{
    return strcmp(*a, *b);
}
```

NAME

ran - random number generator

SYNOPSIS

double ran()

DESCRIPTION

ran returns as its value a random number between 0.0 and 1.0.

NAME

read - read from device or file without buffering

SYNOPSIS

read (fd, buf,bufsize)
int fd, bufsize; char *buf;

DESCRIPTION

read reads characters from a device or disk file which has been previously opened by a call to *open* or *creat.* In most cases, the information is read directly into the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer into which the information is to be placed.

bufsize is the number of characters to be transferred.

If *read* is successful, it returns as its value the number of characters transferred.

If the returned value is zero, then end-of-file has been reached, immediately, with no bytes read.

SEE ALSO

Unbuffered I/O (O), open, close

DIAGNOSTICS

If the operation isn't successful, *read* returns -1 and places a code in the global integer *errno.*

NAME

rename - rename a disk file

SYNOPSIS

```
rename(oldname, newname)        /* non-Unix function */
char *oldname,*newname;
```

DESCRIPTION

rename changes the name of a file.

oldname is a pointer to a character array containing the old file name, and *newname* is a pointer to a character array containing the new name of the file.

If successful, *rename* returns 0 as its value; if unsuccessful, it returns -1.

If a file with the new name already exists, *rename* sets E_EXIST in the global integer *errno* and returns -1 as its value without renaming the file.

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include "stdio.h"

scanf(format [,pointer] ...)
char *format;

fscanf(stream, format [,pointer] ...)
FILE *stream;
char *format;

sscanf(buffer, format [,pointer] ...)
char *buffer, *format;
```

DESCRIPTION

These functions convert a string or stream of text characters, as directed by the control string pointed at by the *format* parameter, and place the results in the fields pointed at by the *pointer* parameters.

The functions get the text from different places:

scanf gets text from the standard input stream, *stdin*;

fscanf gets text from the stream specified in its first parameter, *stream*;

sscanf gets text from the buffer pointed at by its first parameter, *buffer*.

The scan functions are in both *c.lib* and *m.lib*, the difference being that the *c.lib* versions don't support floating point conversions. Hence, if floating point conversion is required, the *m.lib* versions must be used. If floating point conversions aren't required, either version can be used. To use *m.lib*'s version, *m.lib* must be specified before *c.lib* when the program is linked.

The control string pointed at by *format* contains the following 'control items':

* Conversion specifications;
* 'White space' characters (space, tab newline);
* Ordinary characters; that is, characters which aren't part of a conversion specification and which aren't white space.

A scan function works its way through a control string, trying to match each control item to a portion of the input stream or buffer. During the matching process, it fetches characters one at a time from the input. When a character is fetched which isn't appropriate for the control item being matched, the scan function pushes it back into the input stream or buffer and

finishes processing the current control item. This pushing back frequently gives unexpected results when a stream is being accessed by other i/o functions, such as *getc*, as well as the scan function. The examples below demonstrate some of the problems that can occur.

The scan function terminates when it first fails to match a control item or when the end of the input stream or buffer is reached. It returns as its value the number of matched conversion specifications, or EOF if the end of the input stream or buffer was reached.

Matching 'white space' characters

When a white space character is encountered in the control string, the scan function fetches input characters until the first non-white-space character is read. The non-white-space character is pushed back into the input and the scan function proceeds to the next item in the control string.

Matching ordinary characters

If an ordinary character is encountered in the control string, the scan function fetches the next input character. If it matches the ordinary character, the scan function simply proceeds to the next control string item. If it doesn't match, the scan function terminates.

Matching conversion specifications

When a conversion specification is encountered in the control string, the scan function first skips leading white space on the input stream or buffer. It then fetches characters from the stream or buffer until encountering one that is inappropriate for the conversion specification. This character is pushed back into the input.

If the conversion specification didn't request assignment suppression (discussed below), the character string which was read is converted to the format specified by the conversion specification, the result is placed in the location pointed at by the current pointer argument, and the next pointer argument becomes current. The scan function then proceeds to the next control string item.

If assignment suppression was requested by the conversion specification, the scan function simply ignores the fetched input characters and proceeds to the next control item.

Details of input conversion

A conversion specification consists of:

* The character '%', which tells the scan function that it

has encountered a conversion specification;
* Optionally, the assignment suppression character '*';
* Optionally, a 'field width'; that is, a number specifying the maximum number of characters to be fetched for the conversion;
* A conversion character, specifying the type of conversion to be performed.

If the assignment suppression character is present ina conversion specification, the scan function will fetch characters as if it was going to perform the conversion, ignore them, and proceed to the next control string item.

The following conversion characters are supported:

% A single '%' is expected in the input; no assignment is done.

d A decimal integer is expected; the input digit string is converted to binary and the result placed in the *int* field pointed at by the current *pointer* argument;

o An octal integer is expected; the corresponding *pointer* should point to an *int* field in which the converted result will be placed;

x A hexadecimal integer is expected; the converted value will be placed in the *int* field pointed at by the current *pointer* argument;

s A sequence of characters delimited by white space characters is expected; they, plus a terminating null character, are placed in the character array pointed at by the current *pointer* argument.

c A character is expected. It is placed in the *char* field pointed at by the current *pointer*. The normal skip over leading white space is not done; to read a single char after skipping leading white space, use '%1s'. The field width parameter is ignored, so this conversion can be used only to read a single character.

[A sequence of characters, optionally preceded by white space but not terminated by white space is expected. The input characters, plus a terminating null character, are placed in the character array pointed at by the current *pointer* argument. The left bracket is followed by:

* Optionally, a '^' or '~' character;
* A set of characters;
* A right bracket, ']'.

If the first character in the set isn't ^ or -, the set specifies characters which are allowed; characters are fetched from the input until one is read which isn't in the set.

If the first character in the set is ^ or -, the set specifies characters which aren't allowed; characters are fetched from the input until one is read which is in the set.

e A floating point number is expected. The input string is converted to floating point format and stored in the *float* field pointed at by the current *pointer* argument. The input format for floating point numbers consists of an optionally signed string of digits, possibly containing a decimal point, optionally followed by an exponent field consisting of an E or e followed by an optionally signed digit.

The conversion characters d, o, and x can be capitalized or preceded by *l* to indicate that the corresponding *pointer* is to a *long* rather than an *int*. Similarly, the conversion characters e and f can be capitalized or preceded by *l* to indicate that the corresponding *pointer* is to a *double* rather than a *float*.

The conversion characters o, x, and d can be optionally preceded by *h* to indicate that the corresponding *pointer* is to a *short* rather than an *int*. Since *short* and *int* fields are the same in Aztec C, this option has no effect.

SEE ALSO

Standard I/O (O)

EXAMPLES

1. In this program fragment, *scanf* is used to read values for the int x, the float y, and a character string into the char array z:

```
int x; float y; char z[50];
scanf("%d%f%s", &x, &y, z);
```

The input line

```
32 75.36e-1 rufus
```

will assign 32 to x, 7.536 to y, and "rufus " to z. *scanf* will return 3 as its value, signifying that three conversion specifications were matched.

The three input strings must be delimited by 'white space' characters; that is, by blank, tab, and newline characters. Thus, the three values could also be entered on separate

lines, with the white space character newline used to separate the values.

2. This example discusses the problems which may arise when mixing *scanf* and other input operations on the same stream.

   In the previous example, the character string entered for the third variable, z, must also be delimited by white space characters. In particular, it must be terminated by a space, tab, or newline character. The first such character read by *scanf* while getting characters for z will be 'pushed back' into the standard input stream. When another read of stdin is made later, the first character returned will be the white space character which was pushed back.

   This 'pushing back' can lead to unexpected results for programs that read stdin with functions in addition to *scanf*. Suppose that the program in the first example wants to issue a *gets* call to read a line from stdin, following the *scanf* to stdin. *scanf* will have left on the input stream the white space character which terminated the third value read by *scanf*. If this character is a newline, then *gets* will return a null string, because the first character it reads is the pushed back newline, the character which terminates *gets*. This is most likely not what the program had in mind when it called *gets*.

   It is usually unadvisable to mix *scanf* and other input operations on a single stream.

3. This example discusses the behavior of *scanf* when there are white space characters in the control string.

   The control string in the first example was "%d%f%s". It doesn't contain or need any white space, since *scanf*, when attempting to match a conversion specification, will skip leading white space. There's no harm in having white space before the %d, between the %d and %f, or between the %f and %s. However, placing a white space character after the %s can have unexpected results. In this case, *scanf* will, after having read a character string for z, keep reading characters until a non-white-space character is read. This forces the operator to enter, after the three values for x, y, and z, a non-white space character; until this is done, *scanf* will not terminate.

   The programmer might place a newline character at the end of a control string, mistakenly thinking that this will circumvent the problem discussed in example 2. One might think that *scanf* will treat the newline as it would an

ordinary character in the control string; that is, that *scanf* will search for, and remove, the terminating newline character from the input stream after it has matched the z variable. However, this is incorrect, and should be remembered as a common misinterpretation.

4. *scanf* only reads input it can match. If, for the first example, the input line had been

   32 rufus 75.36e-1

   scanf would have returned with value 1, signifying that only one conversion specification had been matched. x would have the value 32, y and z would be unchanged. All characters in the input stream following the 32 would still be in the input stream, waiting to be read.

5. One common problem in using *scanf* involves mismatching conversion specifications and their corresponding arguments. If the first example had declared y to be a double, then one of the following statements would have been required:

   ```
   scanf("%d%lf%s", &x, &y, z);
   ```

   or

   ```
   scanf("%d%F%s", &x, &y, z);
   ```

   to tell *scanf* that the floating point variable was a double rather than a float.

6. Another common problem in using *scanf* involves passing *scanf* the value of a variable rather than its address. The following call to *scanf* is incorrrect:

   ```
   int x; float y; char z[50];
   scanf("%d%f%s", x, y, z);
   ```

   scanf has been passed the value contained in x and y, and the address of z, but it requires the address of all three variables. The "address of" operator, &, is required as a prefix to x and y. Since z is an array, its address is automatically passed to *scanf*, so z doesn't need the & prefix, although it won't hurt if it is given.

7. Consider the following program fragment:

   ```
   int x; float y; char z[50];
   scanf("%2d%f%*d%[1234567890]", &x, &y, z);
   ```

   When given the following input:

   12345 678 90a65

   scanf will assign 12 to x, 345.0 to y, skip '678', and place

the string '90 ' in z. The next call to *getchar* will return 'a'.

NAME

setbuf - assign buffer to a stream

SYNOPSIS

```
#include "stdio.h"

setbuf(stream, buf)
FILE *stream;
char *buf;
```

DESCRIPTION

setbuf defines the buffer that's to be used for the i/o stream *stream*. If *buf* is not a NULL pointer, the buffer that it points at will be used for the stream instead of an automatically allocated buffer. If *buf* is a NULL pointer, the stream will be completely unbuffered.

When *buf* is not NULL, the buffer it points at must contain BUFSIZ bytes, where BUFSIZ is defined in *stdio.h*.

setbuf must be called after the stream has been opened, but before any read or write operations to it are made.

If the user's program doesn't specify the buffer to be used for a stream, the standard i/o functions will dynamically allocate a buffer for the stream, by calling the function *malloc*, when the first read or write operation is made on the stream. Then, when the stream is closed, the dynamically allocated buffer is freed by calling *free*.

SEE ALSO

Standard I/O (O), malloc

NAME

setjmp, longjmp - non-local goto

SYNOPSIS

```
#include "setjmp.h"

setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

These functions are useful for dealing with errors encountered by the low-level functions of a program.

setjmp saves its stack environment in the memory block pointed at by *env* and returns 0 as its value.

longjmp causes execution to continue as if the last call to *setjmp* was just terminating with value *val. val* cannot be zero.

The parameter *env* is a pointer to a block of memory which can be used by *setjmp* and *longjmp.* The block must be defined using the typedef *jmp_buf.*

WARNING

longjmp must not be called without *env* having been initialized by a call to *setjmp.* It also must not be called if the function that called *setjmp* has since returned.

EXAMPLE

In the following example, the function *getall* builds a record pertaining to a customer and returns the pointer to the record if no errors were encountered and 0 otherwise.

getall calls other functions which actually build the record. These functions in turn call other functions, which in turn ...

getall defines, by calling *setjmp,* a point to which these functions can branch if an unrecoverable error occurs. The low level functions abort by calling *longjmp* with a non-zero value.

If a low level function aborts, execution continues in *getall* as if its call to *setjmp* had just terminated with a non-zero value. Thus by testing the value returned by *setjmp getall* can determine whether *setjmp* is terminating because a low level function aborted.

```
#include "setjmp.h"

jmp_buf envbuf; /* environment saved here by setjmp */

getall(ptr)
char *ptr; /* ptr to record to be built */
{
    if (setjmp(envbuf))
        /* a low level function has aborted */
        return 0;
    getfield1(ptr);
    getfield2(ptr);
    getfield3(ptr);
    return ptr;
}
```

Here's one of the low level functions:

```
getsubfld21(ptr)
char *ptr;
{
    ...
    if (error)
        longjmp(envbuf, -1);
    ...
}
```

NAME

trigonometric functions:
sin, cos, tan, cotan, asin, acos, atan, atan2

SYNOPSIS

```
#include <math.h>

double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double cotan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

double atan2(x,y)
double x;
```

DESCRIPTION

sin, *cos*, *tan*, and *cotan* return trigonometric functions of radian arguments.

asin returns the arc sin in the range -pi/2 to pi/2.

acos returns the arc cosine in the range 0 to pi.

atan returns the arc tangent of x in the range -pi/2 to pi/2.

atan2 returns the arc tangent of x/y in the range -pi to pi.

SEE ALSO

Errors (O)

DIAGNOSTICS

If a trig function can't perform the computation, it returns an arbitrary value and sets a code in the global integer *errno*; otherwise, it returns the computed number, without modifying *errno*.

A function will return the symbolic value EDOM if the argument is invalid, and the value ERANGE if the function value can't be computed. EDOM and ERANGE are defined in the file *errno.h*.

The values returned by the trig functions when the computation can't be performed are listed below. The symbolic values are defined in *math.h*.

function	error	f(x)	meaning
sin	ERANGE	0.0	abs(x) > XMAX
cos	ERANGE	0.0	abs(x) > XMAX
tan	ERANGE	0.0	abs(x) > XMAX
cotan	ERANGE	HUGE	0<x< XMIN
cotan	ERANGE	-HUGEi	-XMIN <x <0
cotan	ERANGE	0.0	abs(x) >= XMAX
asin	EDOM	0.0	abs(x) > 1.0
acos	EDOM	0.0	abs(x) > 1.0
atan2	EDOM	0.0	x = y = 0

NAME

sinh, cosh, tanh

SYNOPSIS

```
#include <math.h>

double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;
```

DESCRIPTION

These functions compute the hyperbolic functions of their arguments.

SEE ALSO

Errors (O)

DIAGNOSTICS

If the absolute value of the argument to *sinh* or *cosh* is greater than 348.6, the function sets the symbolic value ERANGE in the global integer *errno* and returns a huge value. This code is defined in the file *errno.h*.

If no error occurs, the function returns the computed value without modifying *errno*.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, index, rindex - string operations

SYNOPSIS

```
char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *index(s, c)
char *s;

char *rindex(s, c)
char *s;
```

DESCRIPTION

These functions operate on null-terminated strings, as follows:

strcat appends a copy of string *s2* to string *s1*. *strncat* copies at most *n* characters. Both terminate the resulting string with the null character (\0) and return a pointer to the first character of the resulting string.

strcmp compares its two arguments and returns an integer greater than, equal, or less than zero, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *strncmp* makes the same comparison but looks at *n* characters at most.

strcpy copies string *s2* to *s1* stopping after the null character has been moved. *strncpy* copies exactly *n* characters: if *s2* contains less than *n* characters, null characters will be appended to the resulting string until *n* characters have been moved; if *s2* contains *n* or more characters, only the first *n* will be moved, and the resulting string will not be null terminated.

strlen returns the number of characters which occur in *s* up to the first null character.

index returns a pointer to the first occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

rindex returns a pointer to the last occurrance of the character *c* in string *s*, or zero if *c* isn't in the string.

NAME

toupper, tolower

SYNOPSIS

toupper(c)

tolower(c)

#include "ctype.h"

_toupper(c)

_tolower(c)

DESCRIPTION

toupper converts a lower case character to upper case: if *c* is a lower case character, *toupper* returns its upper case equivalent as its value, otherwise *c* is returned.

tolower converts an upper case character to lowr case: if *c* is an upper case character *tolower* returns its lower case equivalent, otherwise *c* is returned.

toupper and *tolower* do not require the header file *ctype.h.*

_toupper and *_tolower* are macro versions of *toupper* and *tolower*, respectively. They are defined in *ctype.h.* The difference between the two sets of functions is that the macro versions will sometimes translate non-alphabetic characters, whereas the function versions don't.

NAME

ungetc - push a character back into input stream

SYNOPSIS

```
#include "stdio.h"

ungetc(c, stream)
FILE *stream;
```

DESCRIPTION

ungetc pushes the character *c* back on an input stream. That character will be returned by the next *getc* call on that stream. *ungetc* returns *c* as its value.

Only one character of pushback is guaranteed. EOF cannot be pushed back.

SEE ALSO

Standard I/O (O)

DIAGNOSTICS

ungetc returns EOF (-1) if the character can't be pushed back.

NAME

unlink

SYNOPSIS

unlink(name)
char *name;

DESCRIPTION

unlink erases a file.

name is a pointer to a character array containing the name of the file to be erased.

unlink returns 0 if successful.

DIAGNOSTICS

unlink returns -1 if it couldn't erase the file and places a code in the global integer *errno* describing the error.

NAME

write

SYNOPSIS

write(fd,buf,bufsize)
int fd, bufsize; char *buf;

DESCRIPTION

write writes characters to a device or disk which has been previously opened by a call to *open* or *creat*. The characters are written to the device or file directly from the caller's buffer.

fd is the file descriptor which was returned to the caller when the device or file was opened.

buf is a pointer to the buffer containing the characters to be written.

bufsize is the number of characters to be written.

If the operation is successful, *write* returns as its value the number of characters written.

SEE ALSO

Unbuffered I/O (O) , open, close, read

DIAGNOSTICS

If the operation is unsuccessful, *write* returns -1 and places a code in the global integer *errno*.

STYLE

Chapter Contents

Style ..... style
1. Introduction 3
2. Structured Programming 7
3. Top-down Programming 8
4. Defensive Programming and Debugging 10
5. Things to watch out for 15

Style

This section was written for the programmer who is new to the C language, to communicate the special character of C and the programming practices for which it is best suited. This material will ease the new user's entry into C. It gives meaning to the peculiarities of C syntax, in order to avoid the errors which will otherwise disappear only with experience.

1. Introduction

what's in it for me?

These are the benefits to be reaped by following the methods presented here:

* Reduced debugging times;
* Increased program efficiency;
* Reduced software maintenance burden.

The aim of the responsible programmer is to write straightforward code, which makes his programs more accessible to others. This section on style is meant to point out which programming habits are conducive to successful C programs and which are especially prone to cause trouble.

The many advantages of C can be abused. Since C is a terse, subtle language, it is easy to write code which is unclear. This is contrary to the "philosophy" of C and other structured programming languages, according to which the structure of a program should be clearly defined and easily recognizable.

keep it simple

There are several elements of programming style which make C easier to use. One of these is *simplicity*. Simplicity means *keep it simple*. You should be able to see exactly what your code will do, so that when it doesn't you can figure out why.

A little suspicion can also be useful. The particular "problem areas" which are discussed later in this section are points to check when code "looks right" but does not work. A small omission can cause many errors.

learn the C idioms

C becomes more valuable and more flexible with time. Obviously, elementary problems with syntax will disappear. But more importantly,

C can be described as "idiomatic." This means that certain expressions become part of a standard vocabulary used over and over.

For example,

```
while ((c = getchar()) != EOF)
```

is readily recognized and written by any C programmer. This is often used as the beginning of a loop which gets a character at a time from a source of input. Moreover, the inside set of parentheses, often omitted by a new C programmer, is rarely forgotten after this construct has been used a few times.

be flexible in using the library

The standard library contains a choice of functions for performing the same task. Certain combinations offer advantages, so that they are used routinely. For instance, the standard library contains a function, *scanf*, which can be used to input data of a given format. In this example, the function "scans" input for a floating point number:

```
scanf("%f", &flt_num);
```

There are several disadvantages to this function. An important debit is that it requires a lot of code. Also, it is not always clear how this function handles certain strings of input. Much time could be spent researching the behavior of this function. However, the equivalent to the above is done by the following:

```
flt_num = atof(gets(inp_buf));
```

This requires considerably less code, and is somewhat more straightforward. *gets* puts a line of input into the buffer, "inp_buf," and *atof* converts it to a floating point value. There is no question about what the input function is "looking for" and what it should find.

Furthermore, there is greater flexibility in the second method of getting input. For instance, if the user of the program could enter either a special command ("e" for exit) or a floating point value, the following is possible:

```
gets(inp_buf);
if (inp_buf[0] == 'e')
        exit(0);
flt_num = atof(inp_buf);
```

Here, the first character of input is checked for an "e", before the input is converted to a float.

The relative length of the library description of the *scanf* function is an indication of the problems that can arise with that and related functions.

write readable code

Readability can be greatly enhanced by adhering to what common sense dictates. For instance, most lines can easily accommodate more than one statement. Although the compiler will accept statements which are packed together indiscriminately, the logic behind the code will be lost. Therefore, it makes sense to write no more than one statement per line.

In a similar vein, it is desirable to be generous with whitespace. A blank space should separate the arithmetic and assignment operators from other symbols, such as variable names. And when parentheses are nested, dividing them with spaces is not being too prudent. For example,

```
if((fp=fopen("filename","r")==NULL))
```

is not the same as

```
if ( (fp = fopen("filename", "r")) == NULL )
```

The first line contains a misplaced parenthesis which changes the meaning of the statement entirely. (A file is opened but the file pointer will be null.) If the statement was expanded, as in the second line, the problem could be easily spotted, if not avoided altogther.

use straightforward logical expressions

Conditionals are apt to grow into long expressions. They should be kept short. Conditionals which extend into the next line should be divided so that the logic of the statement can be visualized at a glance. Another solution might be to reconsider the logic of the code itself.

learn the rules for expression evaluation

Keep in mind that the evaluation of an expression depends upon the order in which the operators are evaluated. This is determined from their relative precedence.

Item 7 in the list of "things to watch out for", below, gives an example of what may happen when the evaluation of a boolean expression stops "in the middle". The rule in C is that a boolean will be evaluated only until the value of the expression can be determined.

Item 8 gives a well known example of an "undefined" expression, one whose value is not strictly determined.

In general, if an expression depends upon the order in which it is evaluated, the results may be dubious. Though the result may be strictly defined, you must be certain you know what that definition is.

a matter of taste

There are several popular styles of indentation and placement of the braces enclosing compound statements. Whichever format you

adopt, it is important to be consistent. Indentation is the accepted way of conveying the intended nesting of program statements to other programmers. However, the compiler understands only braces. Making them as visible as possible will help in tracking down nesting errors later.

However much time is devoted to writing readible code, C is low-level enough to permit some very peculiar expressions.

```
/* It is important to insert comments on a regular basis! */
```

Comments are especially useful as brief introductions to function definitions.

In general, moderate observance of these suggestions will lessen the number of "tricks" C will play on you-- even after you have mastered its syntax.

2. Structured Programming

"Structured programming" is an attempt to encourage programming characterized by method and clarity. It stems from the theory that any programming task can be broken into simpler components. The three basic parts are statements, loops, and conditionals. In C, these parts are, respectively, anything enclosed by braces or ending with a semicolon; *for*, *while* and *do-while*; *if-else*.

modularity and block structure

Central to structured programming is the concept of modularity. In one sense, any source file compiled by itself is a module. However, the term is used here with a more specific meaning. In this context, modularity refers to the independence or isolation of one routine from another. For example, a routine such as *main()* can call a function to do a given task even though it does not know how the task is accomplished or what intermediate values are used to reach the final result.

Sections of a program set aside by braces are called "blocks". The "privacy" of C's block structure ensures that the variables of each block are not inadvertently shared by other blocks. Any left brace ({) signals the beginning of a block, such as the body of a function or a *for* loop. Since each block can have its own set of variables, a left brace marks an opportunity to declare a temporary variable.

A function in C is a special block because it is called and is passed control of execution. A function is called, executes and returns. Essentially, a C program is just such a routine, namely, *main*.

A function call represents a task to be accomplished. Program statements which might otherwise appear as several obscure lines can be set aside in a function which satisfies a desired purpose. For instance, *getchar* is used to get a single character from standard input.

When a section of code must be modified, it is simpler to replace a single modular block than it is to delete a section of an unstructured program whose boundaries may be unclear at best. In general, the more precisely a block of program is defined, the more easily it can be changed.

3. Top-down Programming

"Top-down" programming is one method that takes advantage of structured programming features like those discussed above. It is a method of designing, writing, and testing a program from the most general function (i.e., (main()) to the most specific functions (such as getchar()).

All C programs begin with a function called main(). main() can be thought of as a supervisor or manager which calls upon other functions to perform specific tasks, doing little of the work itself. If the overall goal of the program can be considered in four parts (for instance, input, processing, error checking and output), then main() should call at least four other functions.

step one

The first step in the design of a program is to identify what is to be done and how it can be accomplished in a "programmable" way. The *main* routine should be greatly simplified. It needs to call a function to perform the crucial steps in the program. For example, it may call a function, init(), which takes care of all necessary startup initializations. At this point, the programmer does not even need to be certain of all the initializations that will take place in init().

All functions consist of three parts: a parameter list, body, and return value. The design of a function must focus on each of these three elements.

During this first stage of design, each function can be considered a black box. We are concerned only with what goes in and what comes out, not with what goes on inside.

Do not allow yourself to be distracted by the details of the implementation at this point. Flowcharts, pseudocode, decision tables and the like are useful at this stage of the implementation.

A detailed list of the data which is passed back and forth between functions is important and should not be neglected. The interface between functions is crucial.

Although all functions are written with a purpose in mind, it is easy to unwittingly merge two tasks into one. Sometimes, this may be done in the interests of producing a compact and efficient program function. However, the usual result is a bulky, unmanageable function. If a function grows very large or if its logic becomes difficult to comprehend, it should be reduced by introducing additional function calls.

step two

There comes a time when a program must pass from the design stage into the coding stage. You may find the top-down approach to

coding too restrictive. According to this scheme, the smallest and most specific functions would be coded last. It is our nature to tackle the most daunting problems first, which usually means coding the low-level functions.

Whereas the top-down approach is the preferred method for designing software, the bottom-up approach is often the most practical method for writing software. Given a good design, either method of implementation should produce equally good results.

One asset of top-down writing is the ability to provide immediate tests on upper level routines. Unresolved function calls can be satisfied by "dummy" functions which return a range of test values. When new functions are added, they can operate in an environment that has already been tested.

C functions are most effective when they are as mutually independent as is possible. This independence is encouraged by the fact that there is normally only one way into and one way out of a function: by calling it with specific arguments and returning a meaningful value. Any function can be modified or replaced so long as its entry and exit points are consistent with the calling function.

4. Defensive Programming and Debugging

"Defensive programming" obeys the same edict as defensive driving: trust no one to do what you expect. There are two sides to this rule of thumb. Defend against both the possibility of bad data or misuse of the program by the user, and the possibility of bad data generated by bad code.

Pointers, for example, are a prime source of variables gone astray. Even though the "theory" of pointers may be well understood, using them in new ways (or for the first time) requires careful consideration at each step. Pointers present the fewest problems when they appear in familiar settings.

faced with the unknown

When trying something new, first write a few test programs to make sure the syntax you are using is correct. For example, consider a buffer, *str_buf*, filled with null-terminated strings. Suppose we want to print the string which begins at offset *begin* in the buffer. Is this the way to do it?

```
printf("%s", str_buf[begin]);
```

A little investigation shows that str_buf[begin] is a character, not a pointer to a string, which is what is called for. The correct statement is

```
printf("%s", str_buf + begin);
```

This kind of error may not be obvious when you first see it. There are other topics which can be troublesome at first exposure. The promotion of data types within expressions is an example. Even if you are sure how a new construct behaves, it never hurts to doublecheck with a test program.

Certain programming habits will ease the bite of syntax. Foremost among these is simplicity of style. Top-down programming is aimed at producing brief and consequently simple functions. This simplicity should not disappear when the design is coded.

Code should appear as "idiomatic" as possible. Pointers can again provide an example: it is a fact of C syntax that arrays and pointers are one and the same. That is,

```
array[offset]
```

is the same as

```
*(array + offset)
```

The only difference is that an array name is not an lvalue; it is fixed. But mixing the two ways of referencing an object can cause confusion, such as in the last example. Choosing a certain idiom, which is known to behave a certain way, can help avoid many errors in usage.

when bugs strike

The assumption must be that you will have to return to the source code to make changes, probably due to what is called a bug. Bugs are characterized by their persistence and their tendency to multiply rapidly.

Errors can occur at either compile-time or run-time. Compile-time errors are somewhat easier to resolve since they are usually errors in syntax which the compiler will point out.

from the compiler

If the compiler does pick up an error in the source code, it will send an error code to the screen and try to specify where the error occurred. There are several peculiarities about error reporting which should be brought up right away.

The most noticeable of these peculiarities is the number of spurious errors which the compiler may report. This interval of inconsistency is referred to as the compiler's recovery. The safest way to deal with an unusually long list of errors is to correct the first error and then recompile before proceeding.

The compiler will specify where it "noticed" something was wrong. This does not necessarily indicate where you must make a change in the code. The error number is a more accurate clue, since it shows what the compiler was looking for when the error occurred.

if this ever happens to you

A common example of this is error 69: "missing semicolon." This error code will be put out if the compiler is expecting a semicolon when it finds some other character. Since this error most often occurs at the end of a line, it may not be reported until the first character of the following line-- recall that whitespace, such as a newline character, is ignored.

Such an error can be especially treacherous in certain situations. For example, a missing semicolon at the end of a #*include*'d file may be reported when the compiler returns to read input in the original file.

In general, it is helpful to look at a syntax error from the compiler's point of view.

Consider this error:

```
struct structag {
        char c;
        int i;
}

int j;
```

This should generate an error 16: "data type conflict". The arrow in the error message should show that the error was detected right after the "int" in the declaration of *j*. This means that the error has to do with something before that line, since there is nothing illegal about the *int* keyword.

By inspection, we may see that the semicolon is missing from the preceding line. If this fact escapes our notice, we still know that error 16 means this: the compiler found a declaration of the form

[data type] [data type] [symbol name]

where the two data types were incompatible. So while *shortint* is a good data type, *double int* is not. A small intuitive leap leads us to assume that the compiler has read our source as a kind of "struct int" declaration; *struct* is the only keyword preceding the *int* which could have caused this error. Since the compiler is reading the two declarations as a single statement, we must be missing a delimiter.

run-time errors

It takes a bit more ingenuity to locate errors which occur at run-time. In numerical calculations, only the most anomalous results will draw attention to themselves. Other bugs will generate output which will appear to have come from an entirely different program.

A bug is most useful when it is repeatable. Bugs which show up only "sometimes" are merely vexing. They can be caused by a corrupted disk file or a bad command from the user.

When an error can be consistently produced, its source can be more easily located. The nature of an error is a good clue as to its source. Much of your time and sanity will be preserved by setting aside a few minutes to reflect upon the problem.

Which modules are involved in the computation or process? Many possibilities can be eliminated from the start, such as pieces of code which are unrelated to the error.

The first goal is to determine, from a number of possibilities, which module might be the source of the bug.

checking input data

Input to the program can be checked at a low cost. Error checking of this sort should be included on a "routine" basis. For instance, "if ((fp=fopen("file","r"))==NULL)" should be reflex when a file is

opened. Any useful error handling can follow in the body of the *if*.

It is easy to check your data when you first get your hands on it. If an error occurs after that, you have a bug in your program.

printf it

It is useful to know where the data goes awry. One brute force way of tracking down the bug is to insert *printf* statements wherever the data is referenced. When an unexpected value comes up, a single module can be chosen for further investigation.

The printf search will be most effective when done with more refinement. Choose a suspect module. There are only two keys points to check: the entry and return of the function. *printf* the data in question just as soon as the function is entered. If the values are already incorrect, then you will want to make sure the correct data was passed in the function call.

If an incorrect value is returned, then the search is confined to the guilty function. Even if the function returns a good value, you may want to make sure it is handled correctly by the calling function.

If everything seems to be working, jump to the next tricky module and perform another check. When you find a bad result, you will still have to backtrack to discover precisely where the data was spoiled.

function calls

Be aware that data can be garbled in a funtion call. Function parameters must be declared when they are not two byte integers. For instance, if a function is called:

```
fseek(fp, 0, 0);
```

in order to "seek" to the beginning of a file, but the function is defined this way:

```
fseek(fp, offset, origin)
FILE *fp;
long offset;
int origin;
```

there will be unfortunate consequences.

The second parameter is expected to be a *long* integer (four bytes), but what is being passed is a *short* integer (two bytes). In a function call, the arguments are just being pushed onto the stack; when the function is entered, they are pulled off again. In the example, two bytes are being pushed on, but four bytes (whatever four bytes are there) are being pulled off.

The solution is just to make the second parameter a long, with a suffix (0L) or by the cast operator (as in (long)i).

A similar problem occurs when a non-integer return value is not declared in the calling function. For example, if *sqrt* is being called, it must be declared as returning a *double*:

```
double sqrt();
```

This method of debugging demonstrates the usefulness of having a solid design before a function is coded. If you know what should be going into a function and what should be coming out, the process of checking that data is made much simpler.

found it

When the guilty function is isolated, the difficulty of finding the bug is proportional to the simplicity of the code. However, the search can continue in a similar way. You should have a good notion of the purpose of each block, such as a loop. By inserting a *printf* in a loop, you can observe the effect of each pass on the data.

printf's can also point out which blocks are actually being executed. "Falling through" a test, such as an *if* or a *switch*, can be a subtle source of problems. Conditionals should not leave cases untested. An *else*, or a *default* in a *switch*, can rescue the code from unexpected input.

And if you are uncertain how a piece of code will work, it is usually worthwhile to set up small test programs and observe what happens. This is instructional and may reveal a bug or two.

5. Things to Watch Out for

Some errors arise again and again. Not all of them go away with experience. The following list will give you an idea of the kinds of things that can go wrong.

* **missing semicolon or brace**

The compiler will tell you when a missing semicolon or brace has introduced bad syntax into the code. However, often such an error will affect only the logical structure of the program; the code may compile and even execute. When this error is not revealed by inspection, it is usually brought out by a test *printf* which is executed too often or not enough. See compiler error 69.

* **assignment (=) vs comparison (==)**

Since variables are assigned values more often than they are tested for equality, the former operator was given the single keystroke: =. Notice that all the comparison tests with equality are two characters: <=, >= and ==.

* **misplaced semicolon**

When typing in a program, keep in mind that all source lines do not automatically end with a semicolon. Control lines are especially susceptible to an unwanted semicolon:

```
for (i=0; i<100; i++);
        printf("%d",i);
```

This example prints the single number 100.

* **division (/) vs escape sequence (\)**

C definitely distinguishes between these characters. The division sign resides below the question mark on a standard console; the backslash is generally harder to find.

* **character constant vs character string**

Character constants are actually integers equal to the ASCII values of the respective character. A character string is a series of characters terminated by a null character (\0). The appropriate delimiter is the single quote and double quote, respectively.

* **uninitialized variable**

At some point, all variables must be given values before they are used. The compiler will set global and static variables to zero, but automatic variables are guaranteed to contain garbage every time they are created.

* **evaluation of expressions**

For most operations in C, the order of evaluation is rigidly defined; thus, many expressions can be written without lots of parentheses.

However, the order in which unparenthesized expressions are evaluated are not always what you would expect; therefore, it's usually a good idea to use parentheses liberally in expressions where there may be doubt about the order of evaluation (in your mind or in the mind of someone who may later read your program).

For example, the result of the following example is 6:

```
int a = 2, b = 3, c = 4, d;
d = a + b / a * c;
```

The above expression is equivalent to the parenthesized expression *d = a + ((b / a) * c);*. You should probably use some parentheses in this expression, to make its effect clear to yourself and to others.

Consider this example:

```
if ( (c = 0) || (c = 1) )
        printf("%d", c);
```

"1" will be printed; since the first half of the conditional evaluates to zero, the second half must be also evaluated. But in this example:

```
if ( (c = 0) && (c = 1) )
        ;
printf("%d", c);
```

a "0" is printed. Since the first half evaluates to zero, the value of the conditional must be zero, or false, and evaluation stops. This is a property of the logical operators.

* **undefined order of evaluation**

Unfortunately, not all operators were given a complete set of instructions as to how they should be evaluated. A good example is the increment (or decrement) operator. For instance, the following is undefined:

```
i = ++i + --i/++i - i++;
```

How such an expression is evaluated by a particular implementation is called a "side effect." In general, side effects are to be avoided.

* **evaluation of boolean expressions**

Ands, ors and nots invite the programmer to write long conditionals whose very purpose is lost in the code. Booleans should be brief and to the point. Also, the bitwise logical operators must be fully parenthesized. The table in sections 2.12 and 18.1 of *The C Programming Language*, by Kernighan and Ritchie, shows their precedence in relation to other operators.

Here is an extreme example of how a lengthy boolean can be reduced:

```
if ((c = getchar()) != EOF && c >= 'a' && c <= 'z' &&
(c = getchar()) >= '1' && c <= '9')
   printf("good input\n");

if ((c = getchar()) != EOF)
   if (c >= 'a' && c <= 'z')
      if ((c = getchar()) >= '0' && c <= '9')
         printf("good input\n");
```

* **badly formed comments**

The theory of comment syntax is simply that everything occurring between a left /* and a right */ is ignored by the compiler. Nonetheless, a missing */ should not be overlooked as a possible error.

Note that comments cannot be nested, that is

```
/* /*  this will cause an error  */  */
```

And this could happen to you too:

```
/* the rest of this file is ignored until another comment /*
```

* **nesting error**

Remember that nesting is determined by braces and not by indentations in the text of the source. Nested *if* statements merit particular care since they are often paired with an *else*.

* **usage of else**

Every else must pair up with an *if*. When an *else* has inexplicably remained unpaired, the cause is often related to the first error in this list.

* **falling through the cases in a switch**

To maintain the most control over the *cases* in a *switch* statement, it is advisable to end each *case* with a *break*, including the last *case* in the *switch*.

* **strange loops**

The behavior of loops can be explored by inserting *printf* statements in the body of the loop. Obviously, this will indicate if the loop has even been entered at all in course of a run. A counter will show just how many times the loop was executed; a small slip-up will cause a loop to be run through once too often or seldom. The condition for leaving the loop should be doublechecked for accuracy.

* **use of strings**

All strings must be terminated by a null character in memory. Thus, the string, "hello", will occupy a six-element array; the sixth element is ' '. This convention is essential when passing a string to a standard library function. The compiler will append the null character to string constants automatically.

* **pointer vs object of a pointer**

The greatest difficulty in using pointers is being sure of what is needed and what is being used. Functions which take a pointer argument require an address in memory. The best way to ensure that the correct value is being passed is to keep track of what is being pointed to by which pointer.

* **array subscripting**

The first element in a C array has a subscript of zero. The array name without a subscript is actually a pointer to this element. Obviously, many problems can develop from an incorrect subscript. The most damaging can be subscripting out of bounds, since this will access memory above the array and overwrite any data there. If array elements or data stored with arrays are being lost, this error is a good candidate.

* **function interface**

During the design stage, the components of a program should be associated with functions. It is important that the data which is passed among or shared by these functions be explicitly defined in the preliminary design of the program. This will greatly facilitate the coding of the program since the interface between functions must be precise in several respects.

First of all, if the parameters of a function are established, a call can be made without the reservation that it will be changed later. There is less chance that the arguments will be of the wrong type or specified in the wrong order.

A function is given only a private copy of the variables it is passed. This is a good reason to decide while designing the program how functions should access the data they require. You will be able to detail the arguments to be passed in a function call, the global data which the function will alter, the value which the function will return and what declarations will be appropriate-- all without concern for how the function will be coded.

Argument declarations should be a fairly simple matter once these things are known. Note that this declaration list must stand before the left brace of the function body.

The type of the function is the same as the type of the value it returns. Functions must be declared just like any variable. And just like variables, functions will default to type int, that is, the compiler will assume that a function returns an integer if you do not tell it otherwise with a declaration. Thus if function f calls function g which returns a variable of type double, the following declaration is needed:

```
function f()
{
        double g(), bigfloat;

        g(bigfloat);
}
double g(arg)
double arg;
{
        return(arg);
}
```

* **be sure of what a function returns**

You will probably know very well what is returned by a function you have written yourself. But care should be taken when using functions coded by someone else. This is especially true of the standard library functions. Most of the supplied library functions will return an int or a char pointer where you might expect a char. For instance, getchar() returns an int, not a char. The functions supplied by Manx adhere to the UNIX model in all but a few cases.

Of course, the above applies to a function's arguments as well.

* **shared data**

Variables that are declared globally can be accessed by all functions in the file. This is not a very safe way to pass data to functions since once a global variable is altered, there is no returning it to its former state without an elaborate method of saving data. Moreover, global data must be carefully managed; a function may process the wrong variable and consequently inhibit any other function which depends on that data.

Since C provides for and even encourages private data, this definitely should not be a common bug.

COMPILER ERROR MESSAGES

Chapter Contents

Compiler Error Codes ..... err
1. Summary 4
2. Explanations 7
3. Fatal Error Messages 35

Compiler Error Messages

This chapter discusses error messages that can be generated by the compiler. It is divided into three sections: the first summarizes the messages, the second explains them, and the third discusses fatal compiler error messsages.

1. Summary of error codes

No. Interpretation

1: bad digit in octal constant
2: string space exhausted
3: unterminated string
4: internal error
5: illegal type for function
6: inappropriate arguments
7: bad declaration syntax
8: syntax error in typecast
9: array dimension must be constant
10: array size must be positive integer
11: data type too complex
12: illegal pointer reference
13: unimplemented type
14: internal
15: internal
16: data type conflict
17: unsupported data type
18: data type conflict
19: obsolete
20: structure redeclaration
21: missing }
22: syntax error in structure declaration
23: incorrect type for library function (Apprentice C only)
obsolete (other Aztec C compilers)
24: need right parenthesis or comma in arg list
25: structure member name expected here
26: must be structure/union member
27: illegal typecast
28: incompatible structures
29: illegal use of structure
30: missing : in ? conditional expression
31: call of non-function
32: illegal pointer calculation
33: illegal type
34: undefined symbol
35: typedef not allowed here
36: no more expression space
37: invalid expression for unary operator
38: no auto. aggregate initialization allowed
39: obsolete
40: internal
41: initializer not a constant
42: too many initializers

43: initialization of undefined structure
44: obsolete
45: bad declaration syntax
46: missing closing brace
47: open failure on include file
48: illegal symbol name
49: multiply defined symbol
50: missing bracket
51: lvalue required
52: obsolete
53: multiply defined label
54: too many labels
55: missing quote
56: missing apostrophe
57: line too long
58: illegal # encountered
59: macro too long
60: obsolete
61: reference of member of undefined structure
62: function body must be compound statement
63: undefined label
64: inappropriate arguments
65: illegal argument name
66: expected comma
67: invalid else
68: syntax error
69: missing semicolon
70: goto needs a label
71: statement syntax error in do-while
72: 'for' syntax error: missing first semicolon
73: 'for' syntax error: missing second semicolon
74: case value must be an integer constant
75: missing colon on case
76: too many cases in switch
77: case outside of switch
78: missing colon on default
79: duplicate default
80: default outside of switch
81: break/continue error
82: illegal character
83: too many nested includes
84: too many array dimensions
85: not an argument
86: null dimension in array
87: invalid character constant
88: not a structure
89: invalid use of register storage class
90: symbol redeclared

91: illegal use of floating point type
92: illegal type conversion
93: illegal expression type for switch
94: invalid identifier in macro definition
95: macro needs argument list
96: missing argument to macro
97: obsolete
98: not enough arguments in macro reference
99: internal
100: internal
101: missing close parenthesis on macro reference
102: macro arguments too long
103: #else with no #if
104: #endif with no #if
105: #endasm with no #asm
106: #asm within #asm block
107: missing #endif
108: missing #endasm
109: #if value must be integer constant
110: invalid use of : operator
111: invalid use of void expression
112: invalid use function pointer
113: duplicate case in switch
114: macro redefined
115: keyword redefined
116: field width must be > 0
117: invalid 0 length field
118: field is too wide
119: field not allowed here
120: invalid type for field
121: ptr to int conversion
122: ptr & int not same size
123: function ptr & ptr not same size
124: invalid ptr/ptr assignment
125: too many subscripts or indirection on integer

Error codes between 116 and 125 will not occur on Aztec C compilers whose version number is less than 3.

Error codes greater than 200 will occur only if there's something wrong with the compiler. If you get such an error, please send us the program that generated the error.

2. Explanations

1: bad digit in octal constant

The only numerals permitted in the base 8 (octal) counting system are zero through seven. In order to distinguish between octal, hexadecimal, and decimal constants, octal constants are preceded by a zero. Any number beginning with a zero must not contain a digit greater than seven. Octal constants look like this: 01, 027, 003. Hexadecimal constants begin with 0x (e.g., 0x1, 0xAA0, 0xFFF).

2: string space exhausted

The compiler maintains an internal table of the strings appearing in the source code. Since this table has a finite size, it may overflow during compilation and cause this error code. The table default size is about one or two thousand characters depending on the operating system. The size can be changed using the compiler option **-Z**. Through simple guesswork, it is possible to arrive at a table size sufficient for compiling your program.

3: unterminated string

All strings must begin and end with double quotes ("). This message indicates that a double quote has remained unpaired.

4: internal error

This error message should not occur. It is a check on the internal workings of the compiler and is not known to be caused by any particular piece of code. However, if this error code appears, please bring it to the attention of MANX. It could be a bug in the compiler. The release documentation enclosed with the product contains further information.

5: illegal type for function

The type of a function refers to the type of the value which it returns. Functions return an *int* by default unless they are declared otherwise. However, functions are not allowed to return aggregates (arrays or structures). An attempt to write a function such as *struct sam func()* will generate this error code. The legal function types are *char*, *int*, *float*, *double*, *unsigned*, *long*, *void* and a pointer to any type (including structures).

6: error in argument declaration

The declaration list for the formal parameters of a function stands immediately before the left brace of the function body, as shown below. Undeclared arguments default to *int*, though it is usually better practice to declare everything. Naturally, this declaration list may be empty, whether or not the function takes any arguments at all.

No other inappropriate symbols should appear before the left (open) brace.

```
badfunction(arg1, arg2)
shrt arg 1;    /* misspelled or invalid keyword */
double arg 2;
{ /* function body */
}

goodfunction(arg1,arg2)
float arg1;
int arg2;   /* this line is not required */
{ /* function body */
}
```

7: bad declaration syntax

A common cause of this error is the absence of a semicolon at the end of a declaration. The compiler expects a semicolon to follow a variable declaration unless commas appear between variable names in multiple declarations.

```
int i, j;          /* correct */
char c d;          /* error 7 */
char *s1, *s2
float k;           /* error 7 detected here */
```

Sometimes the compiler may not detect the error until the next program line. A missing semicolon at the end of a *#include*'d file will be detected back in the file being compiled or in another *#include* file. This is a good example of why it is important to examine the context of the error rather than to rely solely on the information provided by the compiler error message(s).

8: syntax error in type cast

The syntax of the cast operator must be carefully observed. A common error is to omit a parenthesis:

```
i = 3 * (int number);          /* incorrect usage */
i = 3 * ((int)number);         /* correct usage */
```

9: array dimension must be constant

The dimension given an array must be a constant of type *char*, *int*, or *unsigned*. This value is specified in the declaration of the array. See error 10.

10: array size must be positive integer

The dimension of an array is required to be greater than zero. A dimension less than or equal to zero becomes 1 by default. As can be seen from the following example, specifying a dimension of zero is not the same as leaving the brackets empty.

```
char badarray[0];              /* meaningless */
extern char goodarray[];       /* good */
```

Empty brackets are used when declaring an array that has been defined (given a size and storage in memory) somewhere else (that is, outside the current function or file). In the above example, *goodarray* is external. Function arguments should be declared with a null dimension:

```
func(s1,s2)
char s1[], s2[];
{
    ...
}
```

11: data type too complex

This message is best explained by example:

```
char *******foo;
```

The form of this declaration implies six pointers-to-pointers. The seventh asterisk indicates a pointer to a *char*. The compiler is unable to keep track of so many "levels". Removing just one of the asterisks will cure the error; all that is being declared in any case is a single two-byte pointer. However it is to be hoped that such a construct will never be needed.

12: illegal pointer reference

The type of a pointer must be either *int* or *unsigned*. This is why you might get away with not declaring pointer arguments in functions like *fopen* which return a pointer; they default to *int*. When this error is generated, an expression used as a pointer is of an invalid type:

```
char c;
int var;                       /* any variable */
int varaddress;
varaddress = &var;             /* valid since addresses */
*(varaddress) = 'c';           /* can fit in an int */
*(expression) = 10;            /* in general, expression
                                  must be an int or unsigned */
*c = 'c';                      /* error 12 */
```

13: internal [see error 4]

14: internal [see error 4]

15: storage class conflict

Only automatic variables and function parameters can be specified as *register*.

This error can be caused by declaring a *static register* variable. While structure members cannot be given a storage class at all, function

arguments can be specified only as *register.*

A *register int i* declaration is not allowed outside a function--it will generate error 89 (see below).

16: data type conflict

The basic data types are not numerous, and there are not many ways to use them in declarations. The possibilities are listed below.

This error code indicates that two incompatible data types were used in conjunction with one another. For example, while it is valid to say *long int i,* and *unsigned int j,* it is meaningless to use *double int k* or *float char c.* In this respect, the compiler checks to make sure that *int, char, float* and *double* are used correctly.

data type	*interpretation*	*size(bytes)*
char	character	1
int	integer	2
unsigned/unsigned int	unsigned integer	2
short	integer	2
long/long integer	long integer	4
float	floating point number	4
long float/double	double precision float	8

17: Unsupported data type

This message occurs only when data types are used which are supported by the extended C language, such as the *enum* data type.

18: data type conflict

This message indicates an error in the use of the *long* or *unsigned* data type. *long* can be applied as a qualifier to *int* and *float. unsigned* can be used with *char, int* and *long.*

```
long i;                 /* a long int */
long float d;           /* a double */
unsigned u;             /* an unsigned int */
unsigned char c;
unsigned long l;
unsigned float f;       /* error 18 */
```

19: obsolete

Error codes interpreted as obsolete do not occur in the current version of the compiler. Some simply no longer apply due to the increased adaptability of the compiler. Other error codes have been translated into full messages sent directly to the screen. If you are using an older version of the product and have need of these codes, please contact Manx for information.

20: structure redeclaration

The compiler is able to tell you if a *struct*ure has already been defined. This message informs you that you have tried to redefine a *struct*ure.

21: missing }

The compiler expects to find a comma after each member in the list of fields for a *struct*ure initialization. After the last field, it expects a right (close) brace.

For example, the following program fragment will generate error 21, since the initialization of the structure named 'harry' doesn't have a closing brace:

```
struct sam {
    int bone;
    char license[10];
} harry = {
    1,
    "23-4-1984";
```

22: syntax error in structure declaration

The compiler was unable to find the left (open) brace which follows the tag in a *struct*ure declaration. In the example for error 21, "sam" is the structure tag. A left brace must follow the keyword *struct* if no structure tag is specified.

23: incorrect type for library function (Apprentice C only)

For Apprentice C, this error means that your program has either explicitly or implicitly incorrectly declared the type of a function that's in the run-time system. For example, you will get this error if you call the run-time system function *sqrt* without declaring that it returns a *double*.

23: obsolete (Other Aztec C Compilers)

For Compilers other than Apprentice C, this error should not occur.

24: need right parenthesis or comma

The right parenthesis is missing from a function call. Every function call must have an argument list enclosed by parentheses even if the list is empty. A right parenthesis is required to terminate the argument list.

In the following example, the parentheses indicate that *getchar* is a function rather than a variable.

```
getchar();
```

This is the equivalent of

```
CALL getchar
```

which might be found in a more explicit programming language. In general, a function is recognized as a name followed by a left parenthesis.

With the exception of reserved words, any name can be made a function by the addition of parentheses. However, if a previously defined variable is used as a function name, a compilation error will result.

Moreover, a comma must separate each argument in the list. For example, error 24 will also result from this statement:

```
funccall(arg1, arg2 arg3);
```

25: structure member name expected here

The symbol name following the dot operator or the arrow must be valid. A valid name is a string of alphanumerics and underscores. It must begin with an alphabetic (a letter of the alphabet or an underscore). In the last line of the following example, "(salary)" is not valid because '(' is not an alphanumeric.

```
empptr = &anderson;
empptr->salary = 12000;          /* these three lines */
(*empptr).salary = 12000;        /* are */
anderson.salary = 12000;         /* equivalent */
empptr = &anderson.;             /* error 25 */
empptr-> = 12000;                /* error 25 */
anderson.(salary) = 12000;       /* error 25 */
```

26: must be structure/union member

The defined structure or union has no member with the name specified. If the -S option was specified, no previously defined structure or union has such a member either.

Structure members cannot be created at will during a program. Like other variables, they must be fully defined in the appropriate declaration list. Unions provide for variably typed fields, but the full range of desired types must be anticipated in the union declaration.

27: illegal type cast

It is not possible to cast an expression to a function, a structure, or an array. This message may also appear if a syntax error occurs in the expression to be cast.

```
structure sam { ... } thom;
thom = (struct sam)(expression);          /* error 27 */
```

28: incompatible structures

C permits the assignment of one structure to another. The compiler will ensure that the two structures are identical. Both structures must have the same structure tag. For example:

```
struct sam harry;
struct sam thom;
   ...
harry = thom;
```

29: illegal use of structure

Not all operators can accept a structure as an operand. Also, structures cannot be passed as arguments. However, it is possible to take the address of a structure using the ampersand (&), to assign structures, and to reference a member of a structure using the dot operator.

30: missing : in ? conditional expression

The standard syntax for this operator is:

```
expression ? statement1 : statement2
```

It is not desirable to use ?: for extremely complicated expressions; its purpose lies in brevity and clarity.

31: call of non-function

The following represents a function call:

```
symbol(arg1, arg2, ..., argn);
```

where "symbol" is not a reserved word and the expression stands in the body of a function. Error 31, in reference to the expression above, indicates that "symbol" has been previously declared as something other than a function.

A missing operator may also cause this error:

```
a(b + c);                    /* error 31 */
a * (b + c);                 /* intended */
```

The missing '*' makes the compiler view "a()" as a function call.

32: illegal pointer calculation

Pointers may be involved in three calculations. An integral value can be added to or subtracted from a pointer. Pointers to objects of the same type can be subtracted from one another and compared to one another. (For a formal definition, see Kernighan and Ritchie pp. 188-189.) Since the comparison and subtraction of two pointers is dependent upon pointer size, both operands must be the same size.

33: illegal type

The unary minus (-) and bit complement (~) operators cannot be applied to structures, pointers, arrays and functions. There is no reasonable interpretation for the following:

```
int function();
char array[12];
struct sam { ... } harry;
a = -array;              /* ? */
b = -harry;
c = ~function & WRONG;
```

34: undefined symbol

The compiler will recognize only reserved words and names which have been previously defined. This error is often the result of a typographical error or due to an omitted declaration.

35: typedef not allowed here

Symbols which have been defined as types are not allowed within expressions. The exception to this rule is the use of *sizeof(expression)* and the cast operator. Compare the accompanying examples:

```
struct sam {
    int i;
} harry;
typedef double bigfloat;
typedef struct sam foo;

j = 4 * bigfloat f;      /* error 35 */
k = &foo;                /* error 35 */
x = sizeof(bigfloat);
y = sizeof(foo);         /* good */
```

The compiler will detect two errors in this code. In the first assignment, a typecast was probably intended; compare error 8. The second assignment makes reference to the address of a structure type. However, the structure type is just a template for instances of the structure (such as "harry"). It is no more meaningful to take the address of a structure type than any other data type, as in *&int.*

36: no more expression space

This message indicates that the expression table is not large enough for the compiler to process the source code. It is necessary to recompile the file using the -E option to increase the number of available entries in the expression table. See the description of the compiler in the manual.

37: invalid expression

This error occurs in the evaluation of an expression containing a unary operator. The operand either is not given or is itself an invalid expression.

Unary operators take just one operand; they work on just one variable or expression. If the operand is not simply missing, as in the example below, it fails to evaluate to anything its operator can accept. The unary operators are logical not (!), bit complement (~), increment (++), decrement (--), unary minus (-), typecast, pointer-to (*), address-of (&), and sizeof.

```
if (!) ;
```

38: no auto. aggregate initialization

It is not permitted to initialize automatic arrays and structures. Static and external aggregates may be initialized, but by default their members are set to zero.

```
char array[5] = { 'a', 'b', 'c', 'd' };
function()
{
    static struct sam {
        int bone;
        char license[10];
    } harry = {
        1,
        "123-4-1984"
    };
    char autoarray[2] = { 'f', 'g' };    /* no good */
    extern char array[];
}
```

There are three variables in the above example, only two of which are correctly initialized. The variable "array" may be initialized because it is external. Its first four members will be given the characters as shown. The fifth member will be set to zero.

The structure "harry" is static and may be initialized. Notice that "license" cannot be initialized without first giving a value to "bone". There are no provisions in C for setting a value in the middle of an aggregate.

The variable "autoarray" is an automatic array. That is, it is local to a function and it is not declared to be static. Automatic variables reappear automatically every time a function is called, and they are guaranteed to contain garbage. Automatic aggregates cannot be initialized.

39: obsolete [see error 19]

40: internal [see error 4]

41: initializer not a constant

In certain initializations, the expression to the right of the equals sign (=) must be a constant. Indeed, only automatic and register variables may be initialized to an expression. Such initializations are meant as a convenient shorthand to eliminate assignment statements. The initialization of statics and globals actually occurs at link-time, and not at run-time.

```
{
   int i = 3;
   static int j = (2 + i);       /* illegal */
}
```

42: too many initializers

There were more values found in an initialization than array or structure members exist to hold them. Either too many values were specified or there should have been more members declared in the aggregate definition.

In the initialization of a complex data structure, it is possible to enclose the initializer in a single set of braces and simply list the members, separated by commas. If more than one set of braces is used, as in the case of a structure within a structure, the initializer must be entirely braced.

```
struct {
   struct {
      char array[];
   } substruct;
} superstruct =
```

version 1:

```
   {
      "abcdefghij"
   };
```

version 2:

```
   {
      {
         { 'a','b','c',...,'i','j'}
      }
   };
```

In version 1, the initializers are copied byte-for-byte onto the structure, *superstruct.*

Another likely source of this error is in the initialization of arrays with strings, as in:

```
char array[10] = "abcdefghij";
```

This will generate error 42 because the string constant on the right is null-terminated. The null terminator (' ' or 0x00) brings the size of the initializer to 11 bytes, which overflows the ten-byte array.

43: undefined structure initialization

An attempt has been made to assign values to a structure which has not yet been defined.

```
struct sam {...};
struct dog sam = { 1, 2, 3};  /* error 43 */
```

44: obsolete [see error 19]

45: bad declaration syntax

This error code is an all purpose means for catching errors in declaration statements. It indicates that the compiler is unable to interpret a word in an external declaration list.

46: missing closing brace

All the braces did not pair up at the end of compilation. If all the preceding code is correct, this message indicates that the final closing brace to a function is missing. However, it can also result from a brace missing from an inner block.

Keep in mind that the compiler accepts or rejects code on the basis of syntax, so that an error is detected only when the rules of grammar are violated. This can be misleading. For example, the program below will generate error 46 at the end even though the human error probably occurred in the *while* loop several lines earlier.

As the code appears here, every statement after the left brace in line 6 belongs to the body of the *while* loop. The compilation error vanishes when a right brace is appended to the end of the program, but the results during run time will be indecipherable because the brace should be placed at the end of the loop.

It is usually best to match braces visually before running the compiler. A C-oriented text editor makes this task easier.

```
main()
{
   int i, j;
   char array[80];

   gets(array);
   i = 0;
   while (array[i])  {
      putchar(array[i]);
      i++;
   for ( i=0; array[i];i++) {
      for (j=i + 1; array[j]; j++)  {
         printf("elements %d and %d are ", i, j);
         if (array[i] == array[j])
            printf("the same\n");
         else
            printf("different\n");
      }
      putchar('\n');
   }
}
```

47: open failure on include file

When a file is #*include*d, the compiler will look for it in a default area (see the manual description of the compiler). This message will be generated if the file could not be opened. An open failure usually occurs when the included file does not exist where the compiler is searching for it. Note that a drive specification is allowed in an include statement, but this diminishes flexibility somewhat.

48: illegal symbol name

This message is produced by the preprocessor, which is that part of the compiler which handles lines which begin with a pound sign (#). The source for the error is on such a line. A legal name is a string whose first character is an alphabetic (a letter of the alphabet or an underscore). The succeeding characters may be any combination of alphanumerics (alphabetics and numerals). The following symbols will produce this error code:

```
2nd_time,
dont_do_this!
```

49: multiply defined symbol

This message warns that a symbol has already been declared and that it is illegal to redeclare it. The following is a representative example:

```
int i, j, k, i;          /* illegal */
```

50: missing bracket

This error code is used to indicate the need for a parenthesis, bracket or brace in a variety of circumstances.

51: lvalue required

Only *lvalues* are are allowed to stand on the left-hand side of an assignment. For example:

```
int num;
num = 7;
```

They are distinguished from *rvalues*, which can never stand on the left of an assignment, by the fact that they refer to a unique location in memory where a value can be stored. An *lvalue* may be thought of as a bucket into which an *rvalue* can be dropped. Just as the contents of one bucket can be passed to another, so can an lvalue *y* be assigned to another lvalue, *x*:

```
#define NUMBER  512
x = y;
1024 = z;           /* wrong; l/rvalues are reversed */
NUMBER = x;         /* wrong; NUMBER is still an rvalue */
```

Some operators which require *lvalues* as operands are increment (++), decrement (--), and address-of (&). It is not possible to take the address of a register variable as was attempted in the following example:

```
register int i, j;
i = 3;
j = &i;
```

52: obsolete [see error 19]

53: multiply defined label

On occasions when the goto statement is used, it is important that the specified label be unique. There is no criterion by which the computer can choose between identical labels. If you have trouble finding the duplicate label, use your text editor to search for all occurrences of the string.

54: too many labels

The compiler maintains an internal table of labels which will support up to several dozen labels. Although this table is fixed in size, it should satisfy the requirements of any reasonable C program. C was structured to discourage extravagance in the use of goto's. Strictly speaking, goto statements are not required by any procedure in C; they are primarily recommended as a quick and simple means of exiting from a nested structure.

This error indicates that you should significantly reduce the number of goto's in your program.

55: missing quote

The compiler found a mismatched double quote (") in a *#define* preprocessor command. Unlike brackets, quotes are not paired innermost to outermost, but sequentially. So the first quote is associated with the second, the third with the fourth, and so on. Single quotes (') and double quotes (") are entirely different characters and should not be confused. The latter are used to delimit string constants. A double quote can be included in a string by use of a backslash, as in this example:

```
"this is a string"
"this is a string with an embedded quote: \". "
```

56: missing apostrophe

The compiler found a mismatched single quote or apostrophe (') in a *#define* preprocessor command. Single quotes are paired sequentially (see error 55). Although quotes can not be nested, a quote can be represented in a character constant with a backslash:

```
char c = '\'';          /* c is initialized to
                           single quote  */
```

57: line too long

Lines are restricted in length by the size of the buffer used to hold them. This restriction varies from system to system. However, logical lines can be infinitely long by continuing a line with a backslash-newline sequence. These characters will be ignored.

58: illegal # encountered

The pound sign (#) begins each command for the preprocessor: *#include, #define, #if, #ifdef, #ifndef, #else, #endif, #asm, #endasm, #line and #undef*. These symbols are strictly defined. The pound sign (#) must be in column one and lower case letters are required.

59: macro too long

Macros can be defined with a preprocessor command of the following form:

```
#define [identifier] [substitution text]
```

The compiler then proceeds to replace all instances of "identifier" with the substitution text that was specified by the *#define*.

This error code refers to the substitution text of a macro. Whereas ideally a macro definition may be extended for an arbitrary number of lines by ending each line with a backslash (), for practical purposes the size of a macro has been limited to 255 characters.

60: obsolete [see error 19]

61: reference of member of undefined structure

Occurs only under compilation without the -S option. Consider the following example:

```
int bone;
struct cat {
   int toy;
} manx;
struct dog *samptr;
manx.toy = 1;
bone = samptr->toy;      /* error 61 */
```

This error code appears most often in conjunction with this kind of mistake. It is possible to define a pointer to a structure without having already defined the structure itself. In the example, *samptr* is a structure pointer, but what form that structure ("dog") may take is still unknown. So when reference is made to a member of the structure to which *samptr* points, the compiler replies that it does not even known what the structure looks like.

The *-S* compiler option is provided to duplicate the manner in which earlier versions of UNIX treated structures. Given the example above, it would make the compiler search all previously defined structures for the member in question. In particular, the value of the member "toy" found in the structure "manx" would be assigned to the variable "bone". The -S option is not recommended as a short cut for defining structures.

62: function body must be compound statement

The body of a function must be enclosed by braces, even though it may consist of only one statement:

```
function()
{
   return 1;
}
```

This error can also be caused by an error inside a function declaration list, as in:

```
func(a, b)
int a; chr b;
{
   ...
```

63: undefined label

A *goto* statement is meaningless if the corresponding label does not appear somewhere in the code. The compiler disallows this since it must be able to specify a destination to the computer.

It is not possible to goto a label outside the present function (labels are local to the function in which they appear). Thus, if a label does not exist in the same procedure as its corresponding goto, this message will be generated.

64: inappropriate arguments

When a function is declared (as opposed to defined), it is poor syntax to specify an argument list:

```
function(string)
char *string;
{
   char *func1();              /* correct */
   double func2(x,y);          /* wrong */
   ...
}
```

In this example, function() is being defined, but func1() and func2() are being declared.

65: illegal or missing argument name

The compiler has found an illegal name in a function argument list. An argument name must conform to the same rules as variable names, beginning with an alphabetic (letter or underscore) and continuing with any sequence of alphanumerics and underscores. Names must not coincide with reserved words.

66: expected comma

In an argument list, arguments must be separated by commas.

67: invalid else

An *else* was found which is not associated with an *if* statement. *else* is bound to the nearest *if* at its own level of nesting. So if-else pairings are determined by their relative placement in the code and their grouping by braces.

```
if(...) {
   ...
   if (...) {
      ...
   } else if (...)
      ...
   } else {
      ...
   }
```

The indentation of the source text should indicate the intended structure of the code. Note that the indentation of the if and else-if means only that the programmer wanted both conditionals to be nested at the same level, in particular one step down from the presiding if

statement. But it is the placement of braces that determines this for the compiler. The example above is correct, but probably does not conform to the expectations revealed by the indentation of the else statement. As shown here, the else is paired with the first if, not the second.

68: syntax error

The keywords used in declaring a variable, which specify storage class and data type, must not appear in an executable statement. In particular, all local declarations must appear at the beginning of a block, that is, directly following the left brace which delimits the body of a loop, conditional or function. Once the compiler has reached a non-declaration, a keyword such as *char* or *int* must not lead a statement; compare the use of the casting operator:

```
func()
{
    int i;
    char array[12];
    float k = 2.03;

    i = 0;
    int m;                          /* error 68 */
    j = i + 5;
    i = (int) k;                    /* correct */
    if (i) {
        int i = 3;
        j = i;
        printf("%d",i);
    }
    printf("%d%d\n",i,j);
}
```

This trivial function prints the values 3, 2 and 3. The variable i which is declared in the body of the conditional (if) lives only until the next right brace; then it dies, and the original i regains its identity.

69: missing semicolon

A semicolon is missing from the end of an executable statement. This error code is subject to the same vagaries as its cousin, error 7. It will remain undetected until the following line and is often spuriously caused by a previous error.

70: bad goto syntax

Compare your use of goto with an example. This message says that you did not specify where you wanted to goto with a label:

```
    goto label;
    ...
label:
    ...
```

It is not possible to goto just any identifier in the source code; labels are special because they are followed by a colon.

71: statement syntax error in do-while

The body of a *do-while* may consist of one statement or several statements enclosed in braces. A *while* conditional is required after the body of the loop. This is true even if the loop is infinite, as it is required by the rules of syntax. After typing in a long body, don't forget the *while* conditional.

72: 'for' syntax error: missing first semicolon

This error focuses on another control flow statement, the *for*. The keyword, *for*, must be followed by parentheses. In the parentheses belong three expressions, any or all of which may be null. For the sake of clarity, C requires that the two semicolons which separate the expressions be retained, even if all three expressions are empty.

```
for (;                  /* an infinite loop which does */
    ;                   /* absolutely nothing */
```

Error 72 signifies that the compiler didn't find the first semicolon within the parentheses.

73: 'for' syntax error: missing second semicolon

This error is similar to error 72; it means that the compiler didn't find the second semicolon within the parenthesized expression following the 'for'.

74: case value must be integer constant

Strictly speaking, each value in a *case* statement must be a constant of one of three types: *char*, *int* or *unsigned*. This is similar to the rule for a *switch*ed variable. In the following example, a float must be cast to an int in order to be switched; however, notice that the programmer did not check his case statements. The second case value is invalid, and the code will not compile.

```
float k = 5.0;
switch((int)k)  {
case 4:
   printf("good case value\n");
   break;
case 5.0:
   printf("bad case value\n");
   break;
}
```

The programmer must replace "case 5.0:" with "case 5".

75: missing colon on case

This should be straightforward. If the compiler accepts a case value, a colon should follow it. A semi-colon must not be accidently entered in its place.

76: too many cases in switch

The compiler reserves a limited number of spaces in an internal table for *case* statements. If a program requires more cases than the table initially allows, it becomes necessary to tell the compiler what the table value should be changed to. It is not necessary to know exactly how many are needed; an approximation is sufficient, depending on the requirements of the situation.

77: case outside of switch

The keyword, *case*, belongs to just one syntactic structure, the *switch*. If "case" appears outside the braces which contain a switch statement, this error is generated. Remember that all keywords are reserved, so that they cannot be used as variable names.

78: missing colon

This message indicates that a colon is missing after the keyword, *default*. Compare error 75.

79: duplicate default

The compiler has found more than one *default* in a *switch*. Switch will compare a variable to a given list of values. But it is not always possible to anticipate the full range of values which the variable may take. Nor is it feasible to specify a large number of cases in which the program is not particularly interested.

So C provides for a default case. The default will handle all those values not specified by a case statement. It is analogous to the else companion to the conditional, if. Just as there is one else for every if, only one default case is allowed in a switch statement. However, unlike the else statement, the position of a default is not crucial; a default can appear anywhere in a list of cases.

80: default outside of switch

The keyword, *default*, is used just like *case*. It must appear within the brackets which delimit the switch statement.

81: break/continue error

Break and continue are used to skip the remainder of a loop in order to exit or repeat the loop. Break will also end a switch statement. But when the keywords, *break* or *continue*, are used outside of these contexts, this message results.

82: illegal character

Some characters simply do not make sense in a C program, such as '$' and '@'. Others, for instance the pound sign (#), may be valid only in particular contexts.

83: too many nested includes

#includes can be nested, but this capacity is limited. The compiler will balk if required to descend more than three levels into a nest. In the example given, file D is not allowed to have a #include in the compilation of file A.

```
file A          file B          file C          file D
#include "B"    #include "C"    #include "D"
```

84: too many array dimensions

An array is declared with too many dimensions. This error should appear in conjunction with error 11.

85: not an argument

The compiler has found a name in the declaration list that was not in the argument list. Only the converse case is valid, i.e., an argument can be passed and not subsequently declared.

86: null dimension in array

In certain cases, the compiler knows how to treat multidimensional arrays whose left-most dimensions are not given in its declaration. Specifically, this is true for an extern declaration and an array initialization. The value of any dimension which is not the left-most must be given.

```
extern char array[][12];        /* correct */
extern char badarray[5][];      /* wrong */
```

87: invalid character constant

Character constants may consist of one or two characters enclosed in single quotes, as 'a' or 'ab'. There is no analog to a null string, so '' (two single quotes with no intervening white space) is not allowed. Recall that the special backslash characters (\b, \n, \t etc.) are singular,

so that the following are valid: '\n', '\na', 'a\n'; 'aaa' is invalid.

88: not a structure

Occurs only under compilation without the -S option. A name used as a structure does not refer to a structure, but to some other data type.

```
int i;
i.member = 3;          /* error 88 */
```

89: invalid storage class

A globally defined variable cannot be specified as register. Register variables are required to be local.

90: symbol redeclared

A function argument has been declared more than once.

91: illegal use of floating point type

Floating point numbers can be negated (unary minus), added, subtracted, multiplied, divided and compared; any other operator will produce this error message.

92: illegal type conversion

This error code indicates that a data type conversion, implicit in the code, is not allowed, as in the following piece of code:

```
int i;
float j;
char *ptr;
...
i = j + ptr;
```

The diagram shows how variables are converted to different types in the evaluation of expressions. Initially, variables of type *char* and *short* become *int*, and *float* becomes *double*. Then all variables are promoted to the highest type present in the expression. The result of the expression will have this type also. Thus, an expression containing a *float* will evaluate to a *double*.

hierarchy of types:

```
double <-- float
long
unsigned
int <-- short, char
```

This error can also be caused by an attempt to return a structure, since the structure is being cast to the type of the function, as in:

```
int func()
{
   struct tag sam;
   return sam;
}
```

93: illegal expression type for switch

Only a *char*, *int* or *unsigned* variable can be switched. See the example for error 74.

94: bad argument to define

An illegal name was used for an argument in the definition of a macro. For a description of legal names, see error 65.

95: no argument list

When a macro is defined with arguments, any invocation of that macro is expected to have arguments of corresponding form. This error code is generated when no parenthesized argument list was found in a macro reference.

```
#define   getchar()   getc(stdin)
...
c = getchar;                    /* error 95 */
```

96: missing argument to macro

Not enough arguments were found in an invocation of a macro. Specifically, a "double comma" will produce this error:

```
#define reverse(x,y,z)   (z,y,x)

func(reverse(i,,k));
```

97: obsolete [see error 19]

98: not enough args in macro reference

The incorrect number of arguments was found in an invocation of a previously defined macro. As the examples show, this error is not identical to error 96.

```
#define   exchange(x,y)   (y,x)

func(exchange(i));              /* error 98 */
```

99: internal [see error 4]

100: internal [see error 4]

101: missing close parenthesis on macro reference

A right (closing) parenthesis is expected in a macro reference with arguments. In a sense, this is the complement of error 95; a macro argument list is checked for both a beginning and an ending.

102: macro arguments too long

The combined length of a macro's arguments is limited. This error can be resolved by simply shortening the arguments with which the macro is invoked.

103: #else with no #if

Correspondence between #if and #else is analogous to that which exists between the control flow statements, if and else. Obviously, much depends upon the relative placement of the statements in the code. However, #if blocks must always be terminated by #endif, and the #else statement must be included in the block of the #if with which it is associated. For example:

```
#if ERROR > 0
   printf("there was an error\n");
#else
   printf("no error this time\n");
#endif
```

#if statements can be nested, as below. The range of each #if is determined by a #endif. This also excludes #else from #if blocks to which it does not belong:

```
#ifdef   JAN1
   printf("happy new year!\n");
#if      sick
   printf("i think i'll go home now\n");
#else
   printf("i think i'll have another\n");
#endif
#else
   printf("i wonder what day it is\n");
#endif
```

If the first #endif was missing, error 103 would result. And without the second #endif, the compiler would generate error 107.

104: #endif with no #if

#endif is paired with the nearest #if, #ifdef or #ifndef which precedes it. (See error 103.)

105: #endasm with no #asm

#endasm must appear after an associated #asm. These compiler-control lines are used to begin and end embedded assembly code. This error code indicates that the compiler has reached a #endasm without having found a previous #asm. If the #asm was simply missing, the error list should begin with the assembly code (which are undefined symbols to the compiler).

106: #asm within #asm block

There is no meaningful sense in which in-line assembly code can be nested, so the #asm keyword must not appear between a paired #asm/#endasm. When a piece of in-line assembly is augmented for temporary purposes, the old #asm and #endasm can be enclosed in comments as place-holders.

```
#asm
   /* temporary asm code */
/*  #asm          old beginning  */
   /* more asm code */
#endasm
```

107: missing #endif

A #endif is required for every #if, #ifdef and #ifndef, even if the entire source file is subject to a single conditional compilation. Try to assign pairs beginning with the first #endif. Backtrack to the previous #if and form the pair. Assign the next #endif with the nearest unpaired #if. When this process becomes greatly complicated, you might consider rethinking the logic of your program.

108: missing #endasm

In-line assembly code must be terminated by a #endasm in all cases. #asm must always be paired with a #endasm.

109: #if value must be integer constant

#if requires an integral constant expression. This allows both integer and character constants, the arithmetic operators, bitwise operators, the unary minus (-) and bit complement, and comparison tests.

Assuming all the macro constants (in capitals) are integers,

```
#if DIFF >= 'A'-'a'
#if (WORD &= ~MASK) >> 8
#if MAR | APR | MAY
```

are all legal expressions for use with #if.

110: invalid use of colon operator

The colon operator occurs in two places: 1. following a question mark as part of a conditional, as in (flag ? 1 : 0); 2. following a label inserted by the programmer or following one of the reserved labels, *case* and *default.*

111: illegal use of a void expression

This error can be caused by assigning a *void* expression to a variable, as in this example:

```
void func();
int h;

h = func(arg);
```

112: illegal use of function pointer

For example,

```
int (*funcptr) ();
   ...
funcptr++;
```

funcptr is a pointer to a function which returns an integer. Although it is like other pointers in that it contains the address of its object, it is not suject to the rules of pointer arithmetic. Otherwise, the offending statement in the example would be interpreted as adding to the pointer the size of the function, which is not a defined value.

113: duplicate case in switch

This simply means that, in a *switch* statement, there are two *case* values which are the same. Either the two *cases* must be combined into one, or one of them must be discarded. For instance:

```
switch (c)  {
case NOOP:
   return (0);
case MULT:
   return (x * y);
case DIV:
   return (x / y);
case ADD:
   return (x + y);
case NOOP:
default:
   return;
}
```

The case of NOOP is duplicated, and will generate an error.

114: macro redefined

For example,

```
#define   islow(n)  (n>=0&&n<5)
   ...
#define   islow(n)  (n>=0&&n<=5)
```

The macro, *islow*, is being used to classify a numerical value. When a second definition of it is found, the compiler will compare the new substitution string with the previous one. If they are found to be different, the second definition will become current, and this error code will be produced.

In the example, the second definition differs from the first in a single character, '='. The second definition is also different from this one:

```
#define  islow(n)  n>=0&&n<=5
```

since the parentheses are missing.

The following lines will not generate this error:

```
#define  NULL 0
  ...
#define  NULL 0
```

But these are different from:

```
#define  NULL ' '
```

In practice, this error message does not affect the compilation of the source code. The most recent "revision" of the substitution string is used for the macro. But relying upon this fact may not be a wise habit.

115: keyword redefined

Keywords cannot be defined as macros, as in:

```
#define  int  foo
```

If you have a variable which may be either, for instance, a short or a long integer, there are alternative methods for switching between the two. If you want to compile the variable as either type of integer, consider the following:

```
#ifdef   LONGINT
   long i;
#else
   short i;
#endif
```

Another possibility is through a *typedef*:

```
#ifdef   LONGINT
   typedef  long    VARTYPE;
#else
   typedef  short   VARTYPE;
#endif

VARTYPE i;
```

116: field width must be > 0

A field in a bit field structure can't have a negative number of bits.

117: invalid 0 length field

A field in a bit field structure can't have zero bits.

118: field is too wide

A field in a bit field structure can't have more than 16 bits.

119: field not allowed here

A bit field definition can only be contained in a structure.

120: invalid type for field

The type of a bit field can only be of type *int* of *unsigned int.*

121: ptr/int conversion

The compiler issues this warning message if it must implicitly convert the type of an expression from pointer to *int* or *long*, or vice versa.

If the program explicitly casts a pointer to an *int* this message won't be issued. However, in this case, error 122 may occur.

For example, the following will generate warning 121:

```
char *cp;
int i;
...
i = cp; /* implicit conversion of char * to int */
```

When the compiler issues warning 121, it will generate correct code if the sizes of the two items are the same.

122: ptr & int not same size

If a program explicitly casts a pointer to an *int*, and the sizes of the two items differ, the compiler will issue this warning message. The code that's generated when the converted pointer is used in an expression will use only as much of the least significant part of the pointer as will fit in an *int.*

123: function ptr & ptr not same size

If a program explicitly casts a pointer to a data item to be a pointer to a function, or vice versa, and the sizes of the two pointers differ, the compiler issues this warning message.

If the program doesn't explicitly request the conversion, warning 124 will be issued instead of warning 123.

124: invalid ptr/ptr assignment

If a program attempts to assign one pointer to another without explicitly casting the two pointers to be of the same type, and the types of the two pointers are in fact different, the compiler will issue this warning message.

The compiler will generate code for the assignment, and if the sizes of the two pointers are the same, the code will be correct. But if the

sizes differ, the code may not be correct.

125: too many subscripts or indirection on integer

This warning message is issued if a program attempts to use an integer as a pointer; that is, as the operand of a star operator.

If the sizes of a pointer and an *int* are the same, the generated code will access the correct memory location, but if they don't, it won't.

For example,

```
char c;
long g;
*0x5c=0; /* warning 125, because 0x5c is an int */
c[i]=0;  /* warning 125, because c+i is an int */
g[i]=0;  /* error 12, because g+i is a long */
```

3. Fatal Compiler Error Messages

If the compiler encouters a "fatal" error, one which makes further operation impossible, it will send a message to the screen and end the compilation immediately.

Out of disk space!

There is no room on the disk for the output file of the compiler. Previous disk files will not be overwritten by the compiler's assembly language output. To make room on the disk, it is usually sufficient to remove unneeded files from the disk.

unknown option:

The compiler has been invoked with an option letter which it does not recognize. The manual explicitly states which options the compiler will accept. The compiler will specify the invalid option letter.

duplicate output file

If an output file name has been specified with the -o option and that file already exists on the disk, the compiler will not overwrite it. -O must specify a new file.

too few arguments for -o option

The compiler expected to find the output filename following the "-o", but didn't find it. The output file name must follow the option letter and the name of the file to be compiled must occur last in the command line.

Open failure on input

The input file specified in the command line does not exist on the disk or cannot be opened. A path or drive specification can be included with a filename according to the operating system in use.

No input!

While the compiler was able to open the input file given in the command line, that file was found to be empty.

Open failure on output

The compiler was unable to create an output file. On some systems, this error could occur if a disk's directory is full.

Local table full! (use -L)

The compiler maintains an internal table of the local variables in the source code. If the number of local symbols in use exceeds the available entries in the table at any time during compilation, the compiler will print this message and quit. The default size of the local symbol table (40 entries) can be changed with the -L option for the

compiler. Local variables are those defined within braces, i.e., in a function body or in a compound statement. The scope of a local variable is the body in which it is defined, that is, it is defined until the next right brace at its own nesting level.

Out of memory!

Since the compiler must maintain various tables in memory as well as manipulate source code, it may run out of memory during operation. The more immediate solution is to vary the sizes of the internal tables using the appropriate compiler options. Often, a compilation will require fewer than the default number of entries in a particular table. By reducing the size of that table, memory space is freed up during compile time. The amount of memory used while compiling does not affect the size or content of the assembly or object file output. If this stategy fails to squeeze the compilation into the available memory, the only solution is to divide the source file into modules which can be compiled separately. These modules can then be linked together to produce a single executable file.

INDEX

Order of chapters in manual

System Dependent Chapters

title	*code*
Overview	ov
Tutorial Introduction	tut
The Compiler	cc
The Assembler	as
The Linker	ln
Utility Programs	util
Library Functions Overview: Amiga Information	libov68
Aztec C68k/Amiga Functions	lib68
Technical Information	tech
Debugging Utilities	debug

System Independent Chapters

Overview of Library Functions	libov
System-Independent Functions	lib
Style	style
Compiler Error Messages	err

Index

Index	index

#include files cc.5

A
access lib68.5-6
acos lib.59-60
acvt util.4-5
adding modules after existing modules in a library (lb) util.29
adding modules at beginning or end of a library (lb) util.29-30
adding modules before existing module (lb) util.28
adding modules to a library (lb) util.28
adump util.6
agetc lib.25-26
amiga functions libamiga.3-23
aputc lib.41-42
arcv util.7
asin lib.59-60
assembler directives as.9-14
  bss as.10
  clist as.13
  cseg as.10
  dc as.11
  dcb as.11
  ds as.12
  else as.14
  end as.10
  endm as.13
  entry as.10
  equ as.9
  far as.12
  global as.10
  if as.14
  include as.13
  list as.12
  macro as.13
  mlist as.13
  near as.12
  noclist as.13
  nolist as.12
  nomlist as.13
  public as.10
  reg as.9
assembler operating instructions as.3
assembler options as.6-7
  -c as.6
  -d as.6
  -i as.5-7
  -l as.6,7
  -o as.6
  -zap as.6
  -n as.6
  -s as.6,7
  -v as.6
  -e as.6
assembly-language functions tech.4-11
assert lib68.7
atan lib.59-60
atan2 lib.59-60
atof lib.8
atoi lib.8
atol lib.8
autoindent (z) util.83

B
brk lib68.8
buffering libov.10-11

C
cctemp cc.4
c source file cc.3
calloc lib.31-32
case table cc.16
cbreak libov.19
ceil lib.16
char cc.21
character-oriented input libov.18
clearerr lib.15
close lib.9
closing streams libov.9
cmp util.8
cnm util.9-12
code segmentation cc.11
colon commands (z) util.108
comments style.17
common problems style.15-19

compatablity between Aztec products cc.24
compiler error checking cc.50
compiler error messages err.3-36
compiler operating instructions cc.3
compiler options cc.13-18
-a cc.5,13
-d cc.13,15
-i cc.5,15
-o cc.4,5
-s cc.13,15
-t cc.4,13
-b cc.13
-e cc.13,16
-l cc.13,15
-y cc.13,15
-z cc.13,17
special options for the amiga cc.13
+b cc.13,18
+c cc.10,13
+h cc.13
+d cc.10,13
+i cc.14
+l cc.14,18
+q cc.14
console i/o libov.7,17
cos lib.59-60
cosh lib.61
cotan lib.59-60
creat lib.10
creating a library util.13
creating an assembly language file cc.5
creating an object code file cc.3
crmod libov.20
ctags utility util.99

D
data formats cc.21-22
defensive programming style.10-14
deleting line util.79
deleting modules from a library (lb) util.31
deleting text (z) util.64,66,78-79
device i/o libov.7,15
diff util.13-16
double cc.22
dynamic buffer allocation libov68.5; libov.22

E
echo mode libov.20
editing an existing file (z) util.62
editing another file (z) util.96
embedded assembler source tech.11
environment variables tut.8; cc.4,6; as.5
error checking cc.26
error processing libov.11-12,23
evaluation of expressions style.16
ex-like commands (z) util.89
execl lib68.9
execlp lib68.9
executing system commands util.101
execv lib68.9
execvp lib68.9
exit lib68.11
exiting z util.61
exp lib.12-13
expression evaluation style.5
expression table cc.16
extended pattern matching util.72
extracting modules from a library (lb) util.32-33

F
fabs lib.16

fclose lib.14
fdopen lib.17-19
feof lib.15
ferror lib.15
fexecl lib68.12
fexecv lib68.12
fflush lib.14
fgets lib.27
file comparison utility
util.13
file i/o libov.6,15
file lists util.98
filenames util.95
fileno lib.15
float cc.22
floating point exceptions
cc.22
floor lib.16
fopen lib.17-19
format lib.37-40
fprintf lib.37-40
fputs lib.43
fread lib.20-21
free lib.31-32,56
freopen lib.17-19
frexp lib.22
fscanf lib.49-55
fseek lib.23-24
ftell lib.23-24
ftoa lib.8
function calls and returns
tech.10
function calls style.13-14
fwrite lib.20-21

G
getc lib.25-26
getchar lib.25-26
getenv lib68.14
gets lib.27
getw lib.25-26
global variables cc.21;
tech.9-10
go util.63
grep util.17-22
grep options util.17
matching character strings
util.19
matching repeating characters
util.19
matching single characters
util.18
pattern matching program
util.17
patterns util.18

H
hd util.23
help in lb util.34

I
immediate macro definition (z)
util.85
include environment variable
cc.6; as.5
include search order
cc.5; as.5
index lib.62-63
indirect macro definition (z)
util.86
in-line assembly language
code cc.23
input file as.3
insert commands (z)
util.65,83,106
insert mode (z) util.60,83
inserting text (z) util.66,83
int cc.22
interrupt handlers tech.12
i/o to other devices libov.7
ioctl lib.28; libov.19
isalnum lib.11
isalpha lib.11
isascii lib.11
isatty lib.28
iscntrl lib.11
isdigit lib.11
islower lib.11
isprint lib.11
ispunct lib.11
isspace lib.11
isupper lib.11

L
large code cc.7-12
large data cc.7-12
lb util.24-34
lb arguments util.24
lb options util.24-25
ldexp lib.22
learning c idioms style.3
libraries cc.11; ln.4-7
library table of contents
util.26
line continuation cc.19
line-oriented input
libov.17-18
linker options ln.9-10
-o ln.7,9,10
-l ln.8-10
-f ln.9,10
-t ln.9,11
-w ln.9,11
+o ln.9
+c ln.9,11
+f ln.9,11
-v ln.9,11
linking process ln.4
list object code util.52
listing file as.4
local moves (z) util.74
local symbol table cc.15,16
log10 lib.12-13
log lib.12-13
long cc.22
long names cc.21
longjmp lib.57-58
lseek lib.29-30

M
macros util.85,107
make util.35-51
aborting util.44
built-in rules util.44
command line util.47
logging commands util.44
macro capability util.40-42
makefile syntax util.36,45-46
rules util.38
standard output util.47
starting make util.46
malloc lib.31-32,56
marking util.76,105
memory allocation lib.31-32
memory models cc.7-12
missing semicolon style.15
mixing unbuffered and
standard i/o calls libov.7
mkarcv util.7
mktemp lib68.15
modf lib.22
modularity style.7
moves within c programs (z)
util.75
moving around on the screen (z)
util.74,105
moving blocks of text (z)
util.79
moving modules before an
existing module (lb) util.30
moving modules to the
beginning or end
of a library (lb) util.31
moving modules within a
library (lb) util.30
moving text between files (z)
util.82
moving within a line (z)
util.74
movmem lib.33
mpu symbols cc.24
multi-module programs
cc.11

N
named buffers (z) util.81
near call as.12

O
obd util.52
object file librarian
util.24-34
object module libraries
tech.8
open lib.34-36
opening files libov.6

opening files and devices
libov.6,9
ord util.53
order of evaluation
style.16
order of library modules
ln.5-6
overview of i/o
libov68.3; libov.4

P
paging util.70
perror lib68.17
pointer cc.22
pow lib.12-13
precompiled #include files
cc.6
pre-opened devices
libov.4; libov68.3
printf lib.37-40
program maintenance utility
util.35-51
program organization
tech.4
programmer information
as.8; tech.5
putc lib.41-42
putchar lib.41-42
puterr lib.41-42
puts lib.43
putw lib.41-42

Q
qsort lib.44-45

R
ran lib.46
ram disk, use of tut.8
random i/o libov.6,10;
random number generator
lib.46
raw mode libov.19
read lib.47
readable code style.5
reading files util.96
realloc lib.31-32
rebuilding a library (lb)
util.33
register usage cc.22; tech.11
relocatable object files ln.3
replacing library modules (lb)
util.32
repeat last substitution(&)
util.91
rindex lib.62-63
run-time errors style.12

S
sbrk lib86.11-12
scanf lib.49-55
scdir lib68.18
screen functions lib68.19-20
scrolling in z util.62,66,70
search order of #include
files cc.6
segment size tech.4
segmented code tech.5
segments in memory tech.4
sequential i/o libov.6,10
set util.54
setbuf lib.56
setdate util.55
setjmp lib.57-58
setmem lib.33
setting options for a file
util.93
sg_flags field libov.19
sgtty fields libov.19
shared data style.19
shifting text util.82
short cc.22
sign extension for char
variables cc.24
silence library option (lb)
util.25,33
sin lib.59-60
sinh lib.61
small code cc.7-12
small data cc.7-12
sort an array lib.44
sort object module list
util.53

source dearchiver util.7
special keys util.69
special symbols cc.20
sprintf lib.37-40
sqrt lib.12-13
square root lib.12-13
sscanf lib.49-55
standard i/o libov.9-13
standard i/o functions
libov.12-13
starting and stopping z
util.62,66,92
strcat lib.62-63
strcmp lib.62-63
strcpy lib.62-63
string merging cc.20
string operations lib.62-63
string searching util.63,71
string table cc.17
strlen lib.62-63
strncat lib.62-63
strncmp lib.62-63
strncpy lib.62-63
structure assign cc.19
structure passing cc.19
structured programming
style.7
substitute command (z)
util.90
supported language features
cc.19
swapmem lib.33

T
tags util.98
tan lib.59-60
tanh lib.61
text editor util.57-108
time lib68.21
tiocgetp libov.19
tiocsetp libov.19
tiocsetn libov.19
tmpfile lib68.23
tmpnam lib68.24
tolower lib.64
top-down programming
style.8-9
toupper lib.64
trigonometric functions
lib.59-60

U
unbuffered and standard i/o
calls libov.7
unbuffered i/o libov.14-16
unbuffered i/o to the console
libov.15
unbuffered i/o to non-console
devices libov.17
undoing changes (z) util.82,107
ungetc lib.65
uninitialized variables
style.15
unlink lib.66
using the linker ln.7

V
verbose library option (lb)
util.25,33
verify program assertion
lib68.7
void data type cc.19

W
word movements (z) util.75
write lib.67
writing files (z) util.95
writing programs cc.19
writing machine independent
code cc.23
writing system-independent
programs libov.18

Y
yank (z) util.80,107

Z
z util.57-108
accessing files util.95-100
adjusting the screen

util.77,104
autoindent util.83
changing lines util.79
changing text util.78
colon commands util.108
command summary util.104-108
ctags utility util.99
cursor util.62,66
deleting line util.79
deleting text util.64,66,78-79
display util.104
duplicating blocks of text util.80
editing an existing file util.62
editing another file util.96
ex-like commands util.89
addresses in ex commands util.89
substitute command util.90
repeat last substitution(&) util.91
executing system commands util.101
exiting z util.61
extended pattern matching util.72
file lists util.98
filenames util.95
go util.63
insert commands util.65,83,106
insert mode util.60,83
inserting text util.66,83
local moves util.74
macros util.85,107
immediate macro definition util.85
indirect macro definition util.86
macro wrapping util.87
re-executing macros util.87
marking util.76,105
moves within c programs util.75
moving around on the screen util.74,105
moving blocks of text util.79
moving text between files util.82
moving within a line util.74
named buffers util.81
option codes util.92,102,104
paging util.70
reading files util.96
scrolling util.62,66,70
setting options for a file util.93
shifting text util.82
special keys util.69
starting and stopping z util.62,66,92
string searching util.63,71
tags util.98
undoing changes util.82,107
word movements util.75
writing files util.95
yank util.80,107
z vs vi util.103

Aztec C68K, Version 3.4 for the Amiga
Release Document
February 1987

This release document introduces the features of Aztec C68K, version 3.4 for the Amiga and is divided into the following sections:

1. Product Description
2. New Users
3. Features
4. Amiga Information
5. Known Limitations
6. Bug fixes since version 3.20a
7. Packaging
8. Additional Documentation

1. Product Description

Aztec C68K, version 3.4, allows you to develop programs in the C language that will run on the Amiga.

Aztec C68K, version 3.4, is distributed in two forms: as an update to users who currently have release 3.20a, and to new users who are receiving Aztec C68K for the first time.

If you are receiving Aztec C68K as an update, this package consists of the following:

* A set of disks, which replace your 3.20 version;
* This release document, which contains information on new features, changes, and fixes since version 3.20a, as well as any additional documentation since the manual was printed.

If you are receiving Aztec C68K as a new user, this package consists of the following:

* Disks containing the software;
* A manual that describes the software;
* This release document, which describes any changes that have occurred since the manual was printed.

To acquaint yourself with Aztec C68K, we recommend that you finish reading this release document and then read the Overview and Tutorial chapters of the Aztec C68K manual.

There are three Aztec C68K systems for the Amiga: *Commercial*, *Developer* and *Professional*. Each system's features are a subset of the next higher system's features.

The manual and the documentation that is appended to this release document describes the *Commercial* system's features. If you have the

Professional or *Developer* system and decide later to upgrade to a higher system, we'll just send you disks and you'll be ready to go!

1.1 Three Systems

1.1.1 The *Professional* System

The *Professional* system contains the basics needed to develop C and/or assembly language programs for the Amiga. It consists of the compiler, assembler, linker, libraries and header files. In addition, there are a number of example programs.

1.1.2 The *Developer* System

The *Developer* system contains everything found in the *Professional* system with the following additions:

* The utility programs *make*, *grep*, *diff*, *obd* and *ord* are provided.
* The special math support libraries for the 68881 and for the Manx IEEE emulation are included.
* The powerful and symbolic debugger, DB, is included.
* The *Z* program editor and *ctags* are included.

1.1.3 The *Commercial* System

The *Commercial* system contains everything found in the *Developer* system with the following additions:

* The *Commercial* system contains source to all library functions that are provided with Aztec C68K, whereas the *Developer* system does not.
* The *Commercial* version comes with a year of free updates.

1.2 Support for version 1.2 of the Amiga Workbench

Version 3.4 of Aztec C68k/Amiga now runs under version 1.2 of the Amiga software.

Disk 1 of Aztec C68k/Amiga contains version 1.2 of the Amiga Workbench. To use Aztec C68k/Amiga with the 1.2 Workbench, boot the Amiga with the version 1.2 Kickstart disk and, when prompted for a Workbench disk, insert Aztec C68k/Amiga disk 1.

We recommend that you use Aztec C68k/Amiga with the version 1.2 Workbench. But you can use the version 1.1 Workbench if you want. To do this, you must copy the Aztec C68k/Amiga files over to a version 1.1 Amiga Workbench disk, being careful not to overwrite any of the version 1.1 files with version 1.2 files.

2. New Users!!

The best way to aquaint yourself with our package is to go through the tutorial introduction which is found in the manual. This provides an introduction to your new C programming environment by walking through some of the commands. The next sections you should read in the manual are the ones on the compiler, assembler and the linker which describe in more detail what these programs do and the options that are available.

Another section to reference for C programming is the style chapter in the manual.

This release document serves several purposes. It lists the contents of the disks that are included in the *Packaging* section. Any new features and bugs fixes are also listed. Please note the *Additional Documentation* section that contains some changes to documentation that occurred after the manual was printed.

3. Features

The following is a brief summary of the new features and changes found in the 3.4 version. Full details are contained in an addendum to the appropriate section of the manual in the Additional Documentation section of this release document.

3.1 Readme

Please check the disks to see if there is a *ReadMe* file on them. This file (if one exists) contains important information that was added after the documentation was printed.

3.2 New Compiler Features

This section describes enhancements and changes made to the compiler.

* There is a new code generator which generates smaller and faster code.
* The *32 bit int* option is now fully implemented and supported. Code generated using 32 bit ints is now much smaller.
* The compiler now supports three different floating point formats, Motorola Fast Floating Point (FFP), IEEE double precision emulation, and Motorola 68881 IEEE double precision hardware emulation.
* There is a new option to the compiler which enables stack depth checking code to be generated as part of the function startup sequence.

* There are some new pre-processor manifests. In particular, AZTEC_C is always defined, while __LARGE_DATA and __LARGE_CODE are defined when the appropriate option is used.
* Structure arguments and return values are now correctly handled.
* Enumerated types are now supported.
* The compiler now generates information for use with a source level debugger.
* The INCLUDE environment variable now supports multiple directories by separating the names with '!'.

3.3 New Assembler Features

This section describes enhancements and changes made to the assembler.

* The assembler has undergone major revision and now provides full support for the 68010, 68020 and 68881.
* The assembler squeeze algorithm has been rewritten and is now much faster on large files. The new algorithm is not recursive, so less stack is required.
* Temporary labels are now supported.
* A number of directives have been added. This allows the assembly language header files supplied by Commodore-Amiga to be assembled directly using the Manx assembler.
* The assembly language header files have been included with the package.

3.4 New Linker Features

This section describes enhancements and changes made to the linker.

* The version 3.4 linker supports scatter loading. The object format generated by the assembler had to change to make this work. As a result, **ALL** object modules that were created with version 3.20a must be recompiled and reassembled to be used with the version 3.4 linker and the version 3.4 libraries.
* There are four different models for scatter loading, ranging from one big hunk to every module in its own hunk.
* The new linker will only generate a JMP instruction at the beginning of Hunk 0 if there has been an entry point defined and the entry point is not already at the beginning of Hunk 0. This has several effects. First, the old *crt0.a68* assumed it was being entered by a JSR and will no longer work. Second, it is

now possible to create printer drivers with the linker.

* The linker gives a warning message if a symbol not in the library (i.e. in your program) overrides a symbol in the library. For example, if a global data variable called "Exit" is defined in your program, the linker will display a message when it sees "Exit" defined in the library as well. This helps to prevent hard to find bugs where a call is made to data space or data is stored in code space.
* The linker supports Amiga object format modules and libraries. It will automatically detect that a module is in Amiga format, but it must be told that a file is to be searched as a library.
* The linker will now automatically add a ".o" extension to files that have no extension. It will also check the current directory AND all directories defined in the CLIB environmnet variable. This means that if you want to link with *segload.o*, you can just give the name and the linker will check the current directory and all the CLIB directories.
* The CLIB environment variable now supports multiple directories separated by ';' or '|'.
* The linker supports overlays via segmentation using the '+o' options as defined in the manual.
* The linker now displays module names as they are linked.
* Linking speed has been improved.

3.5 New Features in DB

There is full documentation on DB included with this release document. The major new features are:

* The debugger now supports multiple windows for debugging multiple tasks simultaneously.
* A number of different types of breakpoints are supported including memory change, memory checksum, low memory checksum, and user defined.
* A function trace which displays each function name and arguments as it is called.
* Support for scatter loaded programs.
* Limited 68020/68881 support.
* Automatic window activation under 1.2.
* Input and output redirection.

* Automatic startup files.
* Smart single step mode.
* Startup from CLI or WorkBench.
* Support for setting breakpoints in Manx segmentation (overlays).
* The debugger can now be started **after** a program has generated a DOS software error requester and can show where the error occurred, and in some cases even recover.

3.6 New Z Features

This section describes enhancements to the Z editor.

* The ZOPT environment variable now works correctly.
* The *:s* command is now supported.
* When Z is started, a line number or tag name can be given as an argument.
* The console window is used under 1.2.
* The window can now be resized and Z will automatically adjust.
* The function keys can now be defined.
* Memory is now allocated dynamically.
* Display of full screens disables the cursor and is faster.
* Display can be disabled during macro execution.
* During insert, ^W deletes the previous word.
* A new command, *:fn* searches a "funclist" file and displays the line containing the keyword.
* The editor now looks for "tags" in the current directory and if not found or the keyword is not found, searches the file specified by the environment variable "TAGS".

3.7 New Library Features

This section describes enhancements to the libraries supplied with Aztec C68K.

* Full support for all 1.2 Amiga library functions.
* The Manx *exec()* and *fexec()* functions now work with 1.1 or 1.2.
* The Manx *exec()* and *fexec()* functions support the new PATH command of the CLI.

* There is now a *setenv()* function for creating environment variables.
* Four versions of each library are supplied, 16-bit small code and data, 32-bit small code and data, 16-bit large code and data, and 32-bit large code and data.
* Routines have been added to support interrupts and tasks.
* The startup code now supports creating a window when run from WorkBench.
* All output to the console is now buffered which speeds up display.
* The *ioctl()* function is now supported.
* The *ChkAbort()* function now calls the *__abort()* function.
* Startup argument parsing is now done in the routines *__cli_parse()* and *__wb_parse()*. These can be stubbed for smaller programs.
* Functions have been added for sending a DOS packet and for converting a console from CON: to RAW: and back.
* The debug functions, *kprintf()*, etc. have been added to the library.
* An integer based random number generator function, *rand()*, has been added. There is also a seeding function, *srand()*.
* The character index functions *strchr()* and *strrchr()* have been implemented.

3.8 Other Additions

* A *touch* program has been added to update the modification time of files.
* An object module display program for Manx object modules has been included.
* A lint file declaring most of the Amiga specific functions has been added.

4. Amiga Information

4.1 About 32 bit Integers and Libraries

The Amiga computer, being a true 16 bit machine, can be best utilized by using 16 bit integers. The Aztec compiler for the Amiga does just that. The problem with using 16 bit integers is that the Amiga libraries were written for psuedo 32 bit integer manipulations. This means that all of the Amiga functions that return integers are returning 32 bits when we are expecting only 16 bits. The symptoms

that you will see are various "pointer to int conversion" message errors.

One way to get around this problem is to use the *+L* compiler option. The *+L* option tells the compiler to generate code that uses 32 bit integers. Although this is the easiest way to use the Amiga libraries with Aztec, there will be a slowdown of execution due to the large integer manipulation.

If making all integers 32 bits is not an acceptable solution, then compile your code using the default option which makes integers 16 bits but remember these rules when passing arguments to Amiga functions:

* Pointers are okay.
* Constants that are passed must be cast to a long or have an 'L' at the end.
* Variables that are passed must be either long or be cast to long.
* Macros may be any of the 3 items previously mentioned, but casting never hurts.
* Functions that are documented (i.e. in the Amiga manuals) as returning integers are really returning 32 bit integers. Be sure then that these functions are declared as returning longs. Also any variables that are set to be the return value of these functions must also be declared as longs. It is best to always include the *functions.h* header file which defines the return values.

4.2 32-bit Libraries

If you use the *+L* option with the compiler, make sure that you use the supplied 32-bit versions of the libraries (i.e. c32.lib, m32.lib ...) when the program is linked. For example, if program *foo.c* required 32-bit integers, it would be compiled and linked like this:

```
cc +L foo.c
ln foo.o -lc32
```

4.3 Buffered STDIO

The *stdio* library has been modified to deal with console devices by buffering output to these devices. The buffer can be flushed by one of four means. First, if the buffer fills, the next character will cause the entire buffer to be written to the console device. Second, if a newline ('0) character is detected, the buffer (including the newline) is written out. Third, if a read is made from any console device, all output to consoles are flushed. Fourth and finally, the buffer can be explicitly flushed by using the *fflush()* macro defined in *stdio.h*.

For the most part, all of this will be transparent and speeds console I/O significantly. The only area where it will affect existing or future programs is when a program displays a message without a newline and without asking for a character. For example, a program might display a period for every 10 lines processed. If the periods are to follow each other on the same line, they will not be visible unless the program performs an *fflush()* immediately after printing the character. Otherwise, the entire line will appear when a newline is printed or a character is read from the console.

4.4 Exec functions

The *exec()* and *fexec()* are functions that are only used in the CLI and make use of information that is not supported by Amiga. (It's a lot better under 1.2 though!) These programs work with Amiga Dos 1.1 and 1.2 but it is possible that they may not work with some later version. We will continually update these functions, keep in mind that programs using these may also need future updates.

5. Known Limitations and Problems

5.1 Brk command

There is an error with the *brk* command mentioned in the *lib68* index. It is not a part of this package.

6. Bug Fixes Since Version 3.20a

Bug fixes will be listed by program/file.

6.1 CC

a) Structure arguments and return values work correctly.

b) Using a structure as a pointer and vice-versa no longer corrupts memory.

c) The bug with for(;;) loops not performing the initial check has been fixed.

d) Arrays of floating point numbers can now be indexed correctly.

e) All other known code generator bugs have been fixed.

6.2 AS

a) The squeeze algorithm has been changed to be non-recursive and to dynamically allocate the squeeze table. It is also much faster on large files.

b) The assembler has all new tables which now generate some instructions that were missing before like "abcd".

6.3 LN

a) The linker was almost completely rewritten, and so previous problems don't really apply. See the new linker documentation for new features.

6.4 Z

a) The startup options work. The name of the options file is either taken from the environment variable ZOPT or if ZOPT is not defined, then the name *devs:z.opt* is used.

b) A bug where lines of 3 characters when shifted did not display correctly was fixed.

c) The <INSERT> and <REPLACE> messages weren't being updated correctly.

d) Z supports dynamic memory allocation in 5K chunks.

e) Substitution command (:s) has been implemented.

6.5 CLIB

a) CLIB environment variable supports multiple entries of specifying where libraries may be found using ';' or '!' as a delimiter.

b) The *exec* functions have been changed to work under 1.1 or under 1.2.

c) Library functions in 1.2 are supported. In particular, *DrawEllipse()*, *AreaEllipse()*, the new intuition calls and the expansion architecture calls are also in the library.

d) The *WBenchMsg* variable is initialized by the startup code for compatibility.

e) *strchr()* and *strrchr()* have been added to the library.

f) The *fexec()* routine has been improved and should handle the CLI programs that it had problems with before.

g) A bug in *localtime()* routine was fixed.

h) *fexecl()*, *fexecv()*, *execlp()*, and *execvp()* support the paths available under 1.2.

i) Stdio when accessing a console device is now buffered.

j) New routines that work with 1.2 Beta 6 or later are *set_raw()* and *set_con()*.

k) Large model libraries have been added. They are the same name as the small model ones with the addition of the letter 'l' after the name, but before the '32' if present.

6.6 M.LIB

a) Four floating point libraries support the different floating point formats:

m.lib - Motorola Fast Floating Point
mx.lib - IEEE Double Precision Floating Point Emulation (Manx version)
ma.lib - IEEE Double Precision Floating Point Emulation (Amiga version)
m8.lib - 68881 Floating Point

b) *ran()* and *fabs()* have been added to the math libraries.

6.7 INCLUDE

a) New include files are from the 1.2 final release and have the comments stripped from them.

b) The Amiga assembly language files are included.

6.8 DB

a) Works with any of 68000/68010/68020/68881 processors.

b) The single step command 't' has been added.

c) Improved checking for programs loaded from other directories and non-existent programs.

d) Limited support for disassembling 68881 instructions.

6.9 Miscellaneous

a) The *SETDAT* program has been fixed to match the documentation. There is no need to type all the numbers anymore.

b) *LB* has been fixed.

c) The *DU*, *LS*, and *TOUCH* commands have been added.

7. Packaging

This section describes the files that are provided with Aztec C68K/Amiga. The Professional version of C68k/Amiga is provided on two disks, named *sys1* and *sys2*. The Developer version is provided on *sys1*, *sys2*, plus a third disk, named *sys3*. The Commercial version is provided on *sys1*, *sys2*, *sys3*, plus a fourth disk, named *sys4*.

7.1 Contents of *sys1:*

The root directory of *sys1:* contains the following files and directories:

bin/	directory of Manx programs
c/	directory of CLI programs
devs/	directory of devices
fonts/	directory of fonts
include/	directory of header files
l/	directory of hardware handlers
lib/	directory of libraries and object modules
libs/	directory of Amiga libraries
t/	directory of temporary files
s/	directory containing the startup-sequence
Trashcan/	
.info	
Trashcan.info	
Disk.info	

7.1.1 Contents of *sys1:bin*

as	assembler
cc	Aztec C68K compiler
ln	overlay linker
mclk	
set	initialize and display environment variables
setdat	initialize the date

7.1.2 Contents of *sys1:c*

AddBuffers	Assign
BindDrivers	Break
CD	ChangeTaskPri
Copy	Date
Delete	Dir
DiskChange	Echo

Ed	Edit
Else	EndCLI
EndIf	Execute
FailAt	Fault
FileNote	If
Info	Install
Join	Lab
List	LoadWb
MakeDir	Mount
NewCLI	Path
Prompt	Protect
Quit	Relabel
Rename	Run
Search	SetDate
Skip	Sort
Stack	Status
Type	Version
Wait	Why

7.1.3 Contents of *sys1:devs*

clipboards/
keymaps/
printers/

clipboard.device	Mountlist
narrator.device	parallel.device
printer.device	serial.device
system-configuration	

7.1.3.1 Contents of *sys1:devs/keymaps*

cdn	d
dk	e
f	gb
i	is
n	s
usa0	usa2

7.1.4 Contents of *sys1:fonts*

"intentionally left blank"

7.1.5 Contents of *sys1:lib*

c.lib	Library of standard and Amiga functions
m.lib	Floating point functions, Motorola fast float version
ma.lib	Floating point functions, IEEE/Amiga version
s.lib	Library of screen functions
crt0.o	
segload.o	

7.1.6 Contents of *sys1:libs*

diskfont.library	icon.library
info.library	mathieeedoubbas.library
mathtrans.library	translator.library
version.library	

7.1.7 Contents of *sys1:include*

assert.h	contains assert macro
ctype.h	macro definitions for the 'is...' functions
errno.h	system independent error codes
fcntl.h	unbuffered I/O symbol definitions
functions.h	function definitions
math.h	transcendental functions declarations
obj68k.h	Aztec object file format
setjmp.h	'setjmp' and 'longjmp' declarations
stat.h	stat function declarations
stdio.h	buffered I/O declarations
time.h	time function declarations

clib/	devices/
exec/	graphics/
hardware/	intuition/
libraries/	resources/
workbench/	

7.1.7.1 Contents of *sys1:include/clib*

macros.h

7.1.7.2 Contents of *sys1:include/devices*

audio.h
bootblock.h
clipboard.h
console.h
conunit.h
gameport.h
input.h
inputevent.h
keyboard.h
keymap.h
narrator.h
parallel.h
printer.h
prtbase.h
serial.h
timer.h
trackdisk.h

7.1.7.3 Contents of *sys1:include/exec*

alerts.h
devices.h
errors.h
exec.h
execbase.h
execname.h
interrupts.h
io.h
libraries.h
lists.h
memory.h
nodes.h
ports.h
resident.h
semaphores.h
tasks.h
types.h

7.1.7.4 Contents of *sys1:include/graphics*

clip.h
collide.h
copper.h
display.h
gels.h
gfx.h
gfxbase.h
gfxmacros.h
graphint.h
layers.h
rastport.h
regions.h
sprite.h
text.h
view.h

7.1.7.5 Contents of *sys1:include/hardware*

adkbits.h	blit.h
cia.h	custom.h
dmabits.h	intbits.h

7.1.7.6 Contents of *sys1:include/intuition*

intuition.h	intuitionbase.h

7.1.7.7 Contents of *sys1:include/libraries*

configregs.h	configvars.h
diskfont.h	dos.h
dosextens.h	expansion.h
filehandler.h	mathffp.h
translator.h	

7.1.7.8 Contents of *sys1:include/resources*

cia.h	disk.h
misc.h	potgo.h

7.1.7.9 Contents of *sys1:include/workbench*

icon.h	startup.h
workbench.h	

7.1.8 Contents of *sys1:l*

Disk-Validator	Port-Handler
Ram-Handler	

7.1.9 Contents of *sys1:t*

"intentionally left blank"

7.1.10 Contents of *sys1:s*

.dbinit	Startup-Sequence

7.2 Contents of *sys2:*

The root directory of *sys2:* contains the following files and directories:

asm/
bin/
examples/
lib/
system/
iff/
crt_src/
lint/
examples/

7.2.1 Contents of *sys2:bin*

lb	library maintenance program

7.2.2 Contents of *sys2:examples*

beep.c	boxomatic.c
gadget.c	leo.c
makefile	makeit
mclk.c	menu.c
moire.c	nart.c
oing.c	palette.c
scales.c	sound.c

7.2.3 Contents of *sys2:lib*

cl.lib	Large code & data, 16-bit int version of *c.lib*
ml.lib	" " " " " *m.lib*
mal.lib	" " " " " *ma.lib*
sl.lib	" " " " " *s.lib*
c32.lib	Small code & data, 32-bit int version of *c.lib*
m32.lib	" " " " " *m.lib*
ma32.lib	" " " " " *ma.lib*
s32.lib	" " " " " *s.lib*
cl32.lib	Large code & data, 32-bit int version of *c.lib*
ml32.lib	" " " " " *m.lib*
mal32.lib	" " " " " *ma.lib*
sl32.lib	" " " " " *s.lib*
lcrt0.o	

7.2.4 Contents of *sys2:asm*

libraries/
workbench/
exec/
graphics/
intuition/
resources/
hardware/
devices/

7.2.4.1 Contents of *sys2:asm/libraries*

configregs.i	configvars.i
diskfont.i	dos.i
dosextens.i	dos_lib.i
expansion.i	filehandler.i
translator.i	

7.2.4.2 Contents of *sys2:asm/workbench*

icon.i	startup.i
workbench.i	

7.2.4.3 Contents of *sys2:asm/exec*

ables.i	alerts.i
devices.i	errors.i
exec.i	execbase.i
execname.i	exec_lib.i
initializers.i	interrupts.i
io.i	libraries.i
lists.i	memory.i
nodes.i	ports.i
resident.i	semaphores.i
strings.i	tasks.i
types.i	

7.2.4.4 Contents of *sys2:asm/graphics*

clip.i	copper.i
display.i	gels.i
gfx.i	gfxbase.i
layers.i	rastport.i
regions.i	sprite.i
text.i	view.i

7.2.4.5 Contents of *sys2:asm/intuition*

intuition.i	intuitionbase.i

7.2.4.6 Contents of *sys2:asm/resources*

cia.i	disk.i
misc.i	potgo.i

7.2.4.7 Contents of *sys2:asm/hardware*

adkbits.i	blit.i
cia.i	custom.i
dmabits.i	intbits.i

7.2.4.8 Contents of *sys2:asm/devices*

audio.i	bootblock.i
clipboard.i	console.i
conunit.i	gameport.i
input.i	inputevent.i
keyboard.i	keymap.i
narrator.i	parallel.i
printer.i	prtbase.i
serial.i	timer.i
trackdisk.i	

7.2.5 Contents of *sys2:system*

IconEd	IconEd.info

7.2.6 Contents of *sys2:iff*

compress.c	jiff.c
jiff.h	jiff_save.c
makefile	plop.c
README	uncompress.c

7.2.7 Contents of *sys2:crt_src*

cliparse.c	crt0.a68
vars.c	wbparse.c
_exit.c	_main.c

7.2.8 Contents of *sys2:lint*

manx.c

7.3 Contents of *sys3:*

The root directory of *sys3:* contains the following files and directories:

lib/
bin/
examples/

7.3.1 Contents of *sys3:lib*

m8.lib	Floating point functions, 68881 version
mx.lib	Floating point functions, IEEE/Manx version
m832.lib	Small code & data, 32-bit int version of *m8.lib*
mx32.lib	" " " " " *mx.lib*
m8l.lib	Large code & data, 16-bit int version of *m8.lib*
mxl.lib	" " " " " *mx.lib*
m8l32.lib	Large code & data, 32-bit int version of *m8.lib*
mxl32.lib	" " " " " *mx.lib*

7.3.2 Contents of *sys3:bin*

adump	arcv
avail	cat
cmp	cnm
ctags	db
DB.info	diff
du	grep
hd	ls
make	mkarcv
obd	ord
touch	z

7.3.3 Contents of *sys3:examples*

shell/
vt100/
prof/
getfile/

balls.c	makefile
makeit	setlace.c
speechtoy.c	speechtoy.info
yaboing.c	

7.4 Contents of *sys4:*

The root directory of *sys4:* contains one directory, named *arc.*

7.4.1 Contents of *sys4:arc*

cia.arc	clist.arc
debug.arc	dos.arc
exec.arc	expand.arc
ffp.arc	graphics.arc
icon.arc	intuit.arc
layers.arc	m881.arc
math.arc	ma_ieee.arc
mch68.arc	misc.arc
mx_ieee.arc	potluck.arc
scr.arc	stdio.arc
sysio.arc	time.arc

8. Additional Documentation

This part of the release document contains several different sections of information.

1. Common Problems
2. Examples
3. Documentation Updates

8.1 Common Problems

A common problems chapter is a good place to reference when you are first having a problem.

8.1.1 Programs From BBS's and Magazines Don't Work

Symptom:

After typing in a program from a magazine or using source from a bulleting board or public domain disk, errors occur in compilation or execution.

Solution:

Because Manx Software Systems wishes to make available the utmost flexibility and power to developers using our software, we provide several options for program compilation. In particular, we provide for a choice of 16 or 32 bit ints and whether code and data references are absolute or relative to a base register. Unfortunately, programs written with the 32 bit absolute model in mind may not work correctly in the 16 bit base relative model.

However, a program written for 16 bit base relative WILL work using the 32 bit absolute model. Thus, when dealing with the source to a program that you have received and wish to use, the safest approach is to use the 32 bit absolute model. To simplify the task, the compiler has a special option, *+p*, which automatically enables 32 bit ints, large code and data, and in addition forces the saving of two registers which must be saved if an interrupt routine is being used.

So, the solution is to compile with *+p* and link with the appropriate library. For example:

```
cc +p prog.c
ln prog.o -lml32 -lcl32
```

If, after all this, the program still fails, it's time to look for a bug!

8.1.2 Printing With %f Doesn't Work

Symptom:

Using *printf()* to print floating point numbers with the %f, %e, or %g formats just prints the letter after the %.

Solution:

This means that the program was incorrectly linked. There are two versions of *printf()* and *scanf()* in the libraries. One version knows about floating point numbers, while the other does not. This is done since most programs don't use floating point. It seemed wasteful to force them all to carry the extra overhead of the *atof()* and *ftoa()* routines. These routines would have been included every time the functions *printf()* or *scanf()* were called.

So, the floating point versions of these two routines are kept in the *m.lib* library. When you link, you should type:

```
ln file.o -lm -lc
```

so that the *m.lib* library is searched before the *c.lib* and the correct routine is loaded.

8.1.3 Pointers Don't Print Correctly

Symptom:

A *printf()* statement that works correctly on other machines doesn't work right on the Amiga.

Solution:

On most machines, printing an address or the value of a pointer can be done using a "%d" or a "%x". However, on the Amiga, a pointer is a long value and must be displayed using the "%ld" or "%lx" format.

8.2 Examples

There is an *examples* directory on disk 2 that contains several examples. A *makefile* is also included that has the compiling and linking options for these files. There is also a batch file that can be "Execute"ed. Note that some of these programs must be compiled with the 32 bit integer option and then linked with the 32 bit library.

8.3 Fishdisks

The following information on Amiga software and how to get this software is from Fred Fish. Any questions or problems should be referred to him.

Fred Fish is happy to announce that the AMIGA Freely Redistributable Software Library now contains 45 disks, with more on the way. Sometime in the next month or so he plans to add a "bug fix" disk, which will contain bug fixes for previous disks. Also planned is a "basic only" disk, containing only programs written in basic (icky, but some people still use it).

8.3.1 What's available

There are "do-nothing-useful" examples of various capabilities of the AMIGA, real development tools, editors, languages, games, and other odds & ends. Also included are machine readable form of many of the examples out of the official AMIGA manuals (including the soon to be released ROM Kernel Manual 1.1), received directly from C-A sources.

8.3.2 How to obtain disks

First, check with your local dealers and user groups. Many already have the first eight or so disks. Since these disks can be copied freely, and widespread distribution is encouraged, they propagate out to central distribution points fairly quickly.

If you just can't wait, or can't find copies locally, he is willing to make these disks available for the cost of media, mailing materials, postage, and miscellaneous expenses (like wear and tear on my drives). His goal is to get as much software as possible into the hands of people that can use and enhance it, and make the AMIGA the success it deserves to be.

Each disk contains all source necessary to recreate the executables provided. All programs are currently compiled with the Lattice C compiler. In a very few cases (noted in the description) the code will not compile or run for some reason, but was considered interesting enough to include anyway.

Disks are typically 85 to 95 percent full.

8.3.3 How to order

To order, send a list of the disks you want, and $6 per disk if you want less than 10 disks, $5 per disk if ordering 10 or more disks, to:

Fred Fish
1346 West Tenth Place
Tempe, Az. 85281

(602) 921-1113 (Sorry, he can only return calls collect.)
seismo!noao!mcdsun!fnf

Time and other jobs permitting, all disks will be mailed via first class mail within 5 days of receipt of order. (Tips may help speed the process.)

Feel free to order more the the current number of disks available. Excess funds will be placed "in escrow" (refundable at any time) and drawn against for automatic mailings of future disks as they become available. He hopes to add at least two to four disks per month to the

library. Given that he has a database of about 300Mb of freely distributable software to draw upon, that should be a fairly easy goal to accomplish.

8.3.4 Distribution criteria

To the best of our knowledge, materials in this library are freely redistributable. This means that they have met one or more of the following conditions:

* The materials contains explicit copyright notices permitting redistribution.
* The materials were posted to a publically accessable electronic bulletin board and did not contain any copyright notice. (Such materials will be removed if it is subsequently shown that copyright notices were illegally removed.)
* The materials were posted to a widely disseminated electronic network (such as usenet), thus implying that their author/poster intended them to be freely distributed. This applies only if they contain no notice limiting distribution.
* The materials contain an explicit notice placing them in the public domain. This is not the same as the first condition.

8.4 Documentation Updates

This section contains updates and additions to the documentation in the manual.

8.4.1 New program options

There are four additional options pages, for the *cc*, *as*, *ln*, and *Z* programs that may inserted in your manual at the end of the appropriate section.

8.4.2 New utilities

There is documentation for three new utility programs that can be added to the *util* section of the Aztec C68K manual. The *du* utility displays the number of disk blocks used. The *ls* program displays information about files and directories. This is a new utility that modifies the date of files.

8.4.3 New library functions

There are a number of new library functions that are documented. The descriptions can be added to the *lib68* section of the manual. The functions include:

__abort	__cli__parse	kputchar
KPutChar	kputc	kputch
KPutCh	kputstr	KPutStr
kputs	KPuts	kgetchar
KGetChar	kgetc	KGetCh
kgetch	KMayGetChar	KMayGetCh
KPutFmt	kprintf	KPrintF
KDoFmt	mprintf	KGetNum
kgetnum	dos__packet	geta4
int__start	int__end	ioctl
rand	srand	set__raw
set__con	scdir	scr__beep
scr__bs	scr__tab	scr__lf
scr__cursup	scr__cursrt	scr__cr
scr__clear	scr__home	scr__eol
scr__linsert	scr__ldelete	scr__cinsert
scr__cdelete	scr__curs	freeseg
segload	setenv	__stkchk
__stkover	stat	strchr
strrchr	time	ctime
localtime	gmtime	asctime
__wb__parse		

8.4.4 More technical info

There are some new *tech info* sections that can be added to manual. There is a description of how segmentation has been implemented on the Amiga. There are two sections that deal with interrupts and tasks. A step by step tutorial builds a "hello world" program that can be run from the WorkBench. For those still with a single drive system, a brief suggestion on how to maximize space efficiency. And finally, some documentation on the various floating point formats supported.

8.4.5 Symbolic debugger

This is the final documentation for *db*, the symbolic debugger. It should be placed in the manual following the *tech info* section.

8.4.6 Technical support

At the end of this document is a discussion of the *MANX technical support* and how to make the most out of it. Included are the phone numbers for both the voice lines and the bulletin board system. Also included are problem report forms that may be used if you should encounter a problem that you wish to mail to us. A good place for this information is at the very end of your manual.

New Options for CC

The following options are new to the compiler, *cc*.

+ff This is Motorola Fast Floating Point (FFP), the default format. Computations are done in single precision (32 bit) floating point. The appropriate math library to link when using this format is *m.lib*. Float or double register variables can be declared and will be mapped onto registers D4-D7 as appropriate.

+fi This is the IEEE Double Precision Floating Point Emulation. There is a choice between linking with Manx Aztec C's library (*mx.lib*) or Amiga's library (*ma.lib*). The *ma.lib* math library provides access to the mathieeedoubbas.library. Up to two register variables may be declared with this format using the register pair D4-D5 and D6-D7.

+f8 This is the 68881 Floating Point and the math library *m8.lib* must be linked in. Four register variables may be declared using FP4-FP7.

+m This option enables stack depth checking code to be generated as part of the function startup sequence. It causes a "jsr __stkchk#" to be generated at the beginning of each function. The *stkchk()* routine is in the library and calls a routine called *_stkover()* if it determines that the stack has grown too large or been corrupted. Currently, *_stkover()* prints a message and exits, but can be a user defined routine instead. Since compiling with this option causes the code to be bigger and execute slower, the final version of your program should be compiled without this option.

-n This option enables the compiler to generate source level debugging information. This information is passed as special lines and characters in the assembly language output file. The assembler automatically passes this information through to the Manx object module where it is gathered together by the linker.

Since the source level debugger is not yet available, the default is for the information not to be generated. This will probably change in a later version.

When the information is generated, object modules may increase significantly in size. However, the end program will remain the same size.

+p This option is for super portability. It automatically enables large code and data, 32 bit ints and in addition forces the D2 and D3 registers to be saved. If there is a problem porting a program from another source, use this option along with the cl32.lib library. Note that the cl32.lib library was compiled using this option.

+r This option tells the compiler that it is okay to use the A4 register as a register variable. This option should only be used if you really need the extra register variable and are using the large code and data model for ALL modules of the program and linking with the *cl.lib* or *cl32.lib* library.

Other New Features

* The compiler now pre-defines the name AZTEC__C, which can be used when writing compiler specific code that it is to distributed.
* The compiler will also define the names __LARGE__CODE and __LARGE__DATA when the *+c* and *+d* options are given. These are currently used in some of the header files to switch between an external definition of some hardware addresses and a macro definition with the address hard-coded in.
* The compiler now supports the enumerated data type.
* Structure arguments and return values are now correctly handled.
* The INCLUDE environment variable will now accept the '!' character as a separator between multiple directory names. For example, to specify the C and assembler header files, use:

```
set INCLUDE=df0:include!df0:asm
```

New Features for AS

There are no new options for AS. However, the *-s* option has been removed. This option was used to allocate a larger squeeze table and is no longer needed. See the discussion of the new squeeze algorithm below.

There are a number of new directives and features in this version of the assembler. Each will be described in detail.

New Processor Support

The assembler was partly redesigned and now supports in addition to the MC68000, the MC68010, the MC68020 and the MC68881 instruction sets and addressing modes. The assembler defaults to assuming that only the MC68000 instructions are valid. The *MACHINE* and *MC68881* directives enable and/or disable the additional instructions and addressing modes.

Squeeze Algorithm

The squeeze algorithm was also redesigned. The new algorithm is no longer recursive and thus no longer requires more than a 4K stack. Space for the table is now dynamically allocated, so all instructions should be considered for squeezing. The new algorithm is orders of magnitude faster on large files.

Temporary Labels

Temporary labels of the form *n$*, where *n* consists of decimal digits, are now supported. These labels are in effect till the next non-temporary label is encountered. For example:

```
1$:     move.l (a0)+,(a1)+
        dbra   d0,1$
```

Changes To Macros

A number of changes have been made to the implementation of macros. First, the syntax of the macro definition has been expanded to allow the macro name to be an argument of the MACRO directive or to be taken from a label if present. For example, previously a macro could only be defined as:

```
        macro  addnum
```

Now, however it can still be defined this way or as:

```
addnum          macro
```

Macro arguments can now be referenced by either %n or \n. The %0 or \0 argument refers to the extension on the macro directive when invoked. Macro arguments that contain a space or comma can be

enclosed in bracketing '<' and '>' characters.

When a backslash is followed by the symbol '@', the assembler generates text of the form ".nnn" where *nnn* has a unique value for each invocation of the macro. This is normally used to generate unique labels within a macro.

The symbol NARG is a special assembler symbol which indicates the number of arguments specified when the macro was invoked. Outside of a macro, the value of NARG is 0.

New Operators

The assembler now supports some additional operators:

!	- inclusive or
^	- exclusive or
~	- bitwise not
//	- modulo

New Directives

The assembler supports a number of new directives.

BLANKS

blanks on/off
blanks yes/no
blanks y/n

This directive controls where the assembler will accept blanks or tabs in the operand field of the instruction.

The default setting of *on* allows blanks to be placed between any two complete items. With this setting all comments must be preceded by a ';'.

The blanks *off* setting treats a blank as the end of the operand field.

CNOP

label cnop n1,n2

This directive is used to force alignment on any boundary at a particular offset. The first value, *n1*, is an offset while the second value, *n2*, specifies the alignment to be used as the base of the offset. For example, to align to an even word boundary:

```
cnop  0,2
```

while to align to a long word boundary:

```
cnop  0,4
```

and finally to align to a word beyond a long word boundary:

```
cnop  2,4
```

Note that this will only take effect relative to the beginning of the current modules code or data. Normally, the linker will not align individual modules to long word boundaries. So, for this directive to work, it must either be the first module linked into the program, or else the *+a* option of the linker must be used to force long word alignment of modules.

EQUR

label equr register

This directive allows a register to be referenced by an alternate name. Reference to the new name is made without regard to case.

EVEN

label even

This directive forces alignment to a word (16 bit) boundary.

FAIL

fail

This directive causes the assembler to generate an error for this line. This can be used in macros which detect the incorrect number of arguments and wish to prevent assembly.

FREG

label freg <register list>

This directive is like the REG directive, except that it is used to specify the floating point registers of the MC68881. The list is either composed of the floating point registers *FP0* through *FP7* or of the floating point control registers *FPIAR/FPCR/FPSR*, but not both.

IFC and IFNC

ifc 'string1','string2'
ifnc 'string1','string2'

These conditionals check to see if the two strings are equal. If they are, the *ifc* will assemble the following code, while *ifnc* will skip it.

IFD and IFND

ifd *symbol*
ifnd *symbol*

These conditionals check to see if the specified symbol has been defined or not. If the symbol has been defined, then *ifd* will assemble the following code, while *ifnd* will not.

OTHER IFS

ifeq *absolute_expression*
ifgt *absolute_expression*
ifge *absolute_expression*
ifle *absolute_expression*
iflt *absolute_expression*
ifne *absolute_expression*

These conditionals perform a comparison of the value of the absolute expression to zero. If the specified condition is true, then the following assembly language is processed, otherwise it is skipped.

MACHINE

machine MC68000
machine MC68010
machine MC68020

This directive enables or disables the additional instructions and addressing modes associated with different processors in the MC68000 family.

MC68881

mc68881

This directive enables the MC68881 floating point instructions to be recognized and assembled by the assembler.

SECTION

label *section* *name,CODE*
label *section* *name,DATA*
label *section* *name,BSS*

This directive performs the same functions as the *cseg* and *dseg* directives. The name parameter, if present, is ignored at the current time. The type paramater is used to switch from CODE and back again. If only a name parameter is specified, the type defaults to CODE.

SET

label set expression

This directive assigns the value of the absolute expression to the symbol specified by *label*. This definition is similar to the *EQU* directive, with the exception that this symbol's value can be changed with another *SET* directive.

TTL

ttl title_string

This directive sets the title of the current module being assembled. This directive is implemented for compatibility with other assemblers and has no effect at the current time.

XDEF and XREF

xdef symbol
xref symbol

These directives are used to specify the definition and reference of global symbols. Currently these are both mapped onto the *PUBLIC* directive.

New Options for LN

The following options are new to the linker, *ln*.

+A This option forces each module to be aligned on a long word boundary. For most applications this is not necessary, and will only make the program larger. However, in certain cases, (BCPL pointers again!) it is necessary for long word alignment.

-G, -Q This option tells the linker to generate a source level debugger output file. This file has the same root name as the program output file and the extension ".dbg". This file will be used by the source level debugger once it becomes available.

+Q Disable the module by module display while linking. The new linker displays the names of the various modules as it reads them. This option tells the linker to be quiet.

-M The linker will now complain if a symbol defined in modules not in the library overrides a symbol defined in the library. This happens when someone declares a global variable, for example, index, which may be used by an internal library routine. In this case, when the internal routine calls *index()*, it will call the location defined by the variable *index*. The linker will now warn that the library symbol *index* has been overridden, alerting the user of a potential problem. The *-M* option tells the linker to skip the warnings.

+L This option is a flag that specifies that the following Amiga object modules are really Amiga libraries until the next *+L* option. The linker will automatically detect that a module is in Amiga format. However, since an Amiga object library is simply a concatenation of a number of modules, it is necessary to tell the linker that the following module is a library. Otherwise, the linker will add ALL the modules in the file to the program. For example, if modules am1.o and am2.o are Amiga libraries they can be linked in as follows along with the program "prog.o":

```
ln prog.o +l am1.o am2.o +l -lc
```

+SSS , +SS , +S The linker now supports scatter loading. There are four models supported. The default model with no *+S* option specified is to have all code be in one hunk.

If *+S* is specified, the linker will place each file in a separate hunk. This means that modules concatenated

together into a single file and all modules pulled from a single library will be placed together in the same hunk.

The third model specifed by *+SS* tells the linker to add files to a hunk till the size of the hunk reaches 8K. The linker will then start a new hunk.

The final option, *+SSS*, tells the linker to put every single module in its own individual hunk.

+O[i] The linker now handles segmentation (overlay) using this option. The executable code in the object modules that follow will be placed in code segment *i*. If *i* is not specified, use the first empty segment number. If the segment already exists, append the code to its end. When segments are used, the linker generates a reference to the symbol *.segload* which is defined in a library module, *segload.o*. This module MUST be in the root segment for the program to function properly. The module is also available directly in the *lib* directory.

Segments are loaded into memory as needed and remain in memory until explicitly removed by the program. The program does this by calling the *freeseg()* routine with the address of a function which is in the segment to be unloaded.

For more information, see the new Code Segmentation section of the Technical Information Chapter.

Other New Features

* The new linker will only generate a JMP instruction at the beginning of Hunk 0 if there has been an entry point defined and the entry point is not already at the beginning of Hunk 0. This has several effects. First, the old *crt0.a68* assumed it was being entered by a JSR and will no longer work. Second, it is now possible to create drivers with the linker.

* The linker will now automatically add a ".o" extension to files that have no extension. It will also check the current directory AND all directories defined in the CLIB environment variable. This means that if you want to link with "segload.o", you can just give the name and the linker will check the current directory and all the CLIB directories.

* The CLIB environment variable, used to specify where libraries may be found, now supports multiple entries when they are separated by a ';' or a '!'. For example, if libraries are in both the "df0:lib" and "ram:" directories, then CLIB would be defined like this:

```
set CLIB=df0:lib!ram:!!
```

The null entry at the end means to check the current directory as well.

New Options for Z

This section describes new features in the Z program editor and should be placed at the end of the regular Z documentation.

Z now accepts options as part of the command line.

-tname This option invokes Z and automatically searches the *tags* file for the specified name. If found, the file is opened and the cursor placed on the appropriate line.

-n# This option invokes Z on the first file specified and places the cursor on the requested line number.

New Features

* The new version of Z now makes use, under version 1.2 of the Amiga operating system, of the ability to set a console window into raw mode. As a result, Z will now directly use the console window of the CLI from which it was invoked instead of creating its own window. If Z is run under version 1.1 or if Z is invoked as a background process it will still create its own window.
* The window Z creates or uses is moveable and sizeable. Z will repaint the screen after the window has been resized to reflect the changes in height and width.
* The ZOPT environment variable now works. The name of the startup options file is either taken from the variable ZOPT or if ZOPT is not defined, then the name *devs:z.opt* is used.
* A new flag setting is available, *sm*, which is used to indicate whether or not macros should perform their operation silently. If non-zero, a macro will perform all iterations and redisplay the screen when finished.
* The substitute command, :s, is now implemented as described in the regular Z documentation.
* The tags command, *:ta*, will now search the file pointed to by the environment variable TAGS if the *tags* file doesn't exist or the tag is not found therein. This allows a global tags file for Amiga specific functions and a program specific tags file.
* A new command, *:fn*, has been added. This command takes a string as an argument and searches the file *funclist* for the specified string. If the file does not exist, or the string is not found, a check is made to see if the environment variable FUNCLIST exists. If so, the file indicated by the environment variable is searched as well.

If the string is found in one of the files, the entire line (up to the width of the screen) is displayed that contained the string. This is most useful for displaying the calling sequence of a particular function. In particular, the file *manx.c* in the *lint* directory is set up in this format for all the Amiga library functions. Thus, if FUNCLIST=xxx/manx.c, where xxx is the path to the file, then typing:

:fn AllocMem

would display in the command line area a line like:

void *AllocMem(size,requirements) long size,requirements;{}

This command may also be invoked by placing the cursor on the name to be searched for and typing the ^_ character.

* During insert mode, if a ^W is typed, the previous word typed is deleted.
* Memory is now allocated dynamically in 5K chunks. Memory is not freed until the program exits.

Function Key Strings

With the Amiga version of Z, strings containing up to 19 characters can be associated with function keys and then exectuted when the function key is typed. A separate string is associated with the shifted and unshifted key.

The string operates as though the individual characters were typed from the keyboard. Thus, if typed while in INSERT mode, the characters will be inserted into the text.

To define the string that's associated with a function key, enter a command of the form:

:se xn=string

where n is the number of the function key (with F10 being 0). *x* should be an *f* for unshifted keys and should be an *s* for shifted keys.

For a current list of the strings that are associated with the function keys, type:

:se all

For your convenience, Z, when it starts, associates some commonly used commands with the function keys. This association is listed in the following table. You can of course redefine the function key string association as defined above. In this table, \ *r* stands for the carriage return character.

Function key	Associated String	
	Normal	Shift
F1	:x\r	:q!\r
F2	:^]	:^
F3	:>	@@
F4	:se	:se ts=
F5	:rew\r	:rew!\r
F6	:ta	:ta!
F7	:e #\r	:e! #\r
F8	:e	:e!
F9	:n\r	:n!\r
F10	:w\r	:w!\r

NAME

du

SYNOPSIS

du directory1 [directory2 ...]

DESCRIPTION

The *du* command displays the disk usage information for the specified directory. The command automatically operates upon any subdirectories. The result is specified for each directory in blocks. The blocks from a subdirectory are added to the parent directory's total.

If no argument is specifed, it defaults to the current directory.

NAME

ls

SYNOPSIS

touch file1 [file2 ...]
ls [-blprst] [file pattern | directory ...]

DESCRIPTION

The *ls* program is a utility for listing the contents of directories and providing information about selected files.

Target files may be specified either by specific pathname, pathnames with wild cards in the trailing part of the path or a directory. If no target is specified, the current directory becomes the default.

The output is normally sorted alphabetically in multiple columns with directories displayed in an offsetting color. When directed to a file, though, the output is alphabetical, one name to a line.

There are a number of options which can be specified:

- *-b* This option displays the number of blocks in the file in addition to any other information requested.
- *-l* This option generates additional information for each file listed. The additional information includes the attributes, size and last modification date.
- *-p* This option forces a blank to be output before each file name and a '-' before each directory.
- *-r* This option reverses the order that has otherwise been specified.
- *-s* This option causes the list to be sorted by size, with the largest files first.
- *-t* This option causes the list to be sorted in chrononolgical order, with the most recent files first.

NAME

touch

SYNOPSIS

touch file1 [file2 ...]

DESCRIPTION

touch will cause the modification date of the specified files to be updated to the current time. The files may be specified using the '*' and '?' wild card characters.

Note that the current RAM: handler does not update the modification time when a file is modified.

NAME

_abort - Abort task

SYNOPSIS

_abort()

DESCRIPTION

_abort is called by the function *Chk_Abort* if the flag *Enable_Abort* is non-zero and the control C key signal has been sent to the task. It currently displays a "^C" on the screen and calls *exit(1)*.

_abort allows "^C" aborts to be captured and either ignored or processed in a special way. If *_abort* returns, *Chk_Abort* performs no other action.

For example, the compiler, assembler, and linker now remove any temporary files when ^C is typed. *_abort* calls a cleanup function which deletes the files, prints a message, and then calls *exit*.

NAME

__cli__parse - parse command line

SYNOPSIS

```
extern int __argc, __arg__len;
extern char **__argv, *__arg__lin;

void __cli__parse(pp, alen, aptr)
struct Process *pp;
long alen;
register char *aptr;
```

DESCRIPTION

The startup routine *__main* calls *__cli__parse*, passing to it the following arguments: *pp* points to the program's Process structure; *aptr* points to the command string; and *alen* is the length of the command string.

__cli__parse parses the command string; allocates memory for the parsed information; and sets up pointers to the parsed information in the global variables *__argc*, *__arg__len*, *__argv*, and *__arg__lin*. These variables are used by *__main* when calling *main*. This allocated memory is freed by *__exit*.

__cli__parse is supplied as a separate module. Programs which wish to do custom argument parsing may supply their own routine which will override the one from the library.

It is also possible to produce smaller programs by replacing this function with a null function.

NAME

debug functions for serial port I/O

DESCRIPTION

The following routines are useful for placing debug messages in programs and especially in tasks that have no other way of producing output. These routines talk directly to the serial port hardware bypassing the serial device driver and any interprocess communication.

1. Character output

```
int kputchar(c)
int KPutChar(c)
int kputc(c)
int kputch(c)
int KPutCh(c)
int c;
```

This routine puts out the character passed to the serial port.

Assembler routines can call the label *KPutChar* with the character in register D0.

2. String output

```
void kputstr(cp)
void KPutStr(cp)
void kputs(cp)
void KPutS(cp)
char *cp;
```

This routine puts out the null terminated character string to the serial port using repeated calls to *kputchar*.

Assembler routines can call the label *KPutStr* with the pointer to the string in register A0.

3. Character input

```
int kgetchar()
int KGetChar()
int kgetc()
int KGetCh()
int kgetch()
```

This routine returns a character from the serial port. It calls *KMayGetChar* until a character is received.

Assembler routines can call the label *KGetChar*, the character will be returned in D0.

4. Check for serial port input

```
int KMayGetChar()
int KMayGetCh()
```

This routine checks to see if a character has been received by the serial hardware and returns it if so. If no character is waiting, the routine returns a -1.

Assembler routines can call the label *KMayGetChar*; the value will be returned in D0.

5. Output formatted string

```
KPutFmt(fmt, args)
char *fmt;
unsigned short *args;

kprintf(fmt, args)
KPrintF(fmt, args)
char *fmt;
unsigned short args;
```

These functions operate similar to the standard *printf*, with the following differences: (1) they use the ROM version of the formatter; (2) the output is sent to the serial port; (3) *KPutFmt* is passed the ADDRESS of the arguments, not the arguments themselves.

Assembly language routines can call the label *KPutFmt* with a pointer to the format string in A0 and a pointer to the arguments in A1.

6. Formatted output using ROM routine

```
KDoFmt(fmt, args, func)
char *fmt;
unsigned short *args;
int (*func)();
```

This routine calls the formatter in ROM which will use the function passed to display the formatted text.

Assembly language routines can call the label *KDoFmt* with the pointer to the format string in A0, the pointer to the arguments in A1, and the pointer to the output function in A2.

7. Formatted output to serial port.

```
mprintf(fmt, args)
char *fmt;
unsigned args;
```

This routine calls the Manx *format* routine with the *kputchar* routine used for output, so output is sent to the serial device.

8. Get decimal value from serial port.

```
KGetNum()
kgetnum()
```

This routine reads and echoes a decimal number from the serial device and returns the value.

Assembly language routines can call the label *KGetNum*; the value is returned in D0.

NAME

dos_packet - output packet

SYNOPSIS

```
long
dos_packet(port, type, arg1, arg2, arg3, arg4, arg5, arg6, arg7)
struct MsgPort *port;
long type, arg1, arg2, arg2, arg3, arg4, arg5, arg6, arg7;
```

DESCRIPTION

dos_packet sends a dos packet with the specified type and arguments to the specified *port.*

NAME

geta4 - set up A4

SYNOPSIS

geta4()

DESCRIPTION

geta4 sets up the A4 register which is used as the base for the small code and data models. It need only be called as the very first thing in a newly created task or process.

There is still a potential problem if *getA4* is more than 32K away from the initial task code. In that case, the call to *getA4* will try to use the A4 register to access the *getA4* routine. In this case, bite the bullet and compile the module containing the initial task code with the large code flag (+c).

NAME

int_start & int_end - routines for interrupt handlers

SYNOPSIS

int_start()
int_end()

DESCRIPTION

These two routines are used at the beginning and end of a C routine that is to be used as an interrupt handler. They save some extra registers and set up the A4 register for the small code and data model. They are equivalent to adding the following lines to the beginning and end of the interrupt routine:

```
#asm
        movem.l   d2/d3/a4,-(sp)
        jsr       __geta4#
#endasm

#asm
        movem.l   (sp)+,d2/d3/d4
#endasm
```

NAME

ioctl - Amiga version of ioctl

SYNOPSIS

```
#include <sgtty.h>

ioctl(fd, cmd, stty)
struct sgttyb *stty;
```

DESCRIPTION

The Amiga version of *ioctl* performs only one function: it can set or reset RAW mode on the selected I/O console device. *ioctl* is only valid under Version 1.2 of the Amiga operating system.

NAME

srand & rand - random number generators

SYNOPSIS

```
int rand()

void srand(seed)
unsigned int seed;
```

DESCRIPTION

These two functions provide a random number generator which is not floating point based. *srand* provides a starting seed for the generator. *rand* returns a random integer value.

NAME

set_raw & set_con - set console window characteristics

SYNOPSIS

set_raw()

set_con()

DESCRIPTION

These functions change the operating characteristics of the console window associated with the current process. *set_raw* switches the operation to that of a RAW: type console window. *set_con* switches operation to the normal CON: console window.

If the window is already in the requested state, no action is performed.

Note that this function will only work under version 1.2 of the Amiga operating system.

NAME

scdir -- return the name of the next file matching pattern

SYNOPSIS

```
char * scdir(pat)
char *pat;
```

DESCRIPTION

scdir is a function which permits the user to perform wild card expansion on file name patterns using native Amiga facilities.

When *scdir* is called with a pattern, it returns a pointer to a static area containing the null terminated name of the next file which matches the pattern or zero if no more files match the pattern. Since the area containing the name is statically allocated, the name will be overwritten by subsequent calls to *scdir*.

Patterns have two wildcard characters. The asterisk (*) matches any number of characters. The question mark (?) matches a single character.

EXAMPLE

```
main()
{
        char *sav[100];
        register char  *pat;
        register int i;

        /* We're looking for all c files on the current directory */

        pat = "*.c";
        i = 0;

        while ((ptr = scdir(pat)) && i < 100) {

                sav[i] = malloc(strlen(ptr)+1);
                strcpy(sav[i++], ptr);

        }
        /* rest of program */
}
```

NAME

screen manipulation functions:
scr_beep, scr_bs, scr_tab, scr_lf,
scr_cursup, scr_cursrt, scr_cr,
scr_clear, scr_home, scr_curs, scr_eol,
scr_linsert, scr_ldelete,
scr_cinsert, scr_cdelete

SYNOPSIS

scr_beep()

scr_bs()

scr_tab()

scr_lf()

scr_cursup()

scr_cursrt()

scr_cr()

scr_clear()

scr_home()

scr_eol()

scr_linsert()

scr_ldelete()

scr_cinsert()

scr_cdelete()

scr_curs(lin, col)
int lin, col;

DESCRIPTION

These functions can be called by command programs to manipulate screens of text. For example, there are functions to clear the screen, position the cursor, and insert and delete characters and lines.

These functions can be used in conjunction with the normal standard i/o and unbuffered i/o functions to display characters on the console.

A program that calls these functions must access the console using the Aztec console driver; that is, it must have been linked with *shcroot* or *mixcroot.*

scr_beep rings the keyboard bell.

scr_bs moves the cursor back one character space, without modifying the character that was backspaced over.

scr_tab moves the cursor right one tab stop.

scr_lf moves the cursor down one line, scrolling if at the bottom of the screen.

scr_cursup moves the cursor up without changing its column location.

scr_cursrt moves the cursor right one character space, without modifying the character that was spaced over.

scr_cr causes a carriage return.

scr_clear clears the screen and homes the cursor.

scr_home homes the cursor to the upper left hand corner of the screen.

scr_curs moves the cursor to the line and column specified by the *lin* and *col* parameters, respectively.

scr_eol erases the line at which the cursor is located, from the current cursor positon to the end of the line.

scr_linsert inserts a blank line at the cursor location, moving the lines below the cursor down one line.

scr_ldelete deletes the line at the cursor location, moving the lines below the cursor up one line and placing a blank line at the bottom of the screen.

scr_cinsert inserts a space at the cursor location, shifting right one character the characters in the line which are on the right of the cursor.

scr_cdelete deletes the character at the cursor location, shifting left one character the characters in the line which are on the right of the cursor.

NAME

segload & freeseg - segment load & unload

SYNOPSIS

```
void freeseg(func)
int (*func)();

void segload(func)
int (*func)();
```

DESCRIPTION

freeseg() is used with segmented (overlay) programs to unload a segment that is no longer needed but is occupying memory. The parameter *func* is the name of any function in the segment.

segload() is used with segmented (overlay) programs to force a segment into memory without actually calling any of the functions in the segment. Normally the segment is automatically loaded when a function in the segment is called from another segment. The parameter *func* is the name of any function in the segment.

This can be used by a program starting up that detects a reasonable amount of free memory and that wishes to put all the program load delay at the beginning of the program. It is also used by the debugger, DB, to force segments into memory so that breakpoints can be set.

NAME

setenv

SYNOPSIS

setenv(name, buf)
char *name, *buf;

DESCRIPTION

This function will add or delete environment variables to or from the pseudo-library used to implement the environment. *name* is a pointer to the environment variable to be changed and *buf* is a pointer to the new value. If *buf* points to a null string, the variable *name* will be removed from the environment.

If the environment does not already exist, it will be created.

NAME

_stkchk & _stkover - stack support functions

SYNOPSIS

void _stkchk()

void _stkover()

DESCRIPTION

_stkchk is an assembly language routine which is called at the beginning of every function which has been compiled with the stack checking option (+m). It attempts to determine if the current stack is below the base which is initialized by the startup code. It also checks to see if a code word placed at the bottom of the stack has been corrupted. If either event has not occurred, control is returned to the function, otherwise, the *_stkover* function is called.

_stkover is called by *_stkchk* when a stack overflow has occurred. The routine supplied in the library resets the stack pointer to the top of the stack area and displays the following message on standard output:

Stack overflow!

It then calls the _exit() routine.

NAME

stat

SYNOPSIS

```
stat(name, buf)
char *name, *buf;
```

DESCRIPTION

stat returns the attribute byte, date and time, and size of the file *name*. This information is returned in *buf*, which has the following format:

```
struct stat {
    char st_attr;
    long st_mtime;
    long st_size;
    long st_rsize;
};
```

This structure, and the meaning of the bits in the attribute and time fields are defined in the header file *stat.h*, and in the TIME section.

name can optionally specify the full pathname where the file is located.

ERRORS

stat returns -1 if it fails, after setting a code in the global integer *errno*. The Errors section of the Library Overview chapter describes these codes.

NAME

strchr & strrchr - find a character in a string

SYNOPSIS

```
char *strchr(s,c)
char *s; int c;

char *strrchr(s,c)
char *s; int c;
```

DESCRIPTION

strchr returns a pointer to the first occurrence of the character *c* in string *s*, or NULL if *c* isn't in the string. *c* can be the null character.

strrchr is like *strchr*, except that it returns a pointer to the last occurrence of *c* in *s*, rather than the first.

SEE ALSO

strchr and *strrchr* are the ANSI standard equivalents of the non-standard functions *index* and *rindex*, respectively. The only difference between them is that the *str...* functions can find a null character, while the *index* functions can't.

NAME

time, ctime, localtime, gmtime, asctime

SYNOPSIS

```
long time(tloc)
long *tloc;

char *ctime(clock)
long *clock;

#include "time.h"

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;
```

DESCRIPTION

time returns the date and time, which it gets from the operating system. The other functions convert the date and time, which are passed as arguments, to another format.

time returns the current date and time packed into a long int. If its argument *tloc* is non-null, the return value is also stored in the field pointed at by the argument. The format of the value returned by *time* is described below.

ctime, *localtime*, and *gmtime* convert a date and time pointed at by their argument, which is in a format such as returned by *time*, to another format:

ctime converts the time to a 26-character ASCII string of the form

Mon Apr 30 10:04:52 1984\n\0

localtime and *gmtime* unpack the date and time into a structure and return a pointer to it. The structure, named *tm*, is described below and defined in the header file *time.h*.

asctime converts a date and time pointed at by its argument, which is in a structure such as returned by *localtime* and *gmtime*, to a 26-character ASCII string in the same form as returned by *ctime*.

The long int returned by *time* and passed to *ctime*, *localtime*, and *gmtime* contains the number of seconds since January 1, 1980.

The structure returned by *localtime* and *gmtime*, and passed to *asctime*, has the following format:

```
struct tm {
    short tm_sec;  /* seconds */
    short tm_min; /* minutes */
    short tm_hour;/* hours */
    short tm_mday;/* day of the month */
    short tm_mon; /* month */
    short tm_year; /* year since 1900 */
    short tm_wday;/* day of the week (0 = Sunday */
    short tm_yday;/* day of year */
    short tm_isdst;/* not used */
    short tm_hsec; /* hundredths of seconds */
}
```

NAME

__wb_parse - open standard I/O window

SYNOPSIS

```
void __wb_parse(pp, wbm)
register struct Process *pp;
struct WBStartup *wbm;
```

DESCRIPTION

__wb_parse() is called from the *__main()* routine and is used to open a window for standard I/O to use. The window is actually defined by setting the ToolType, "WINDOW", to the desired window specification. If this is not required, this routine may be replaced by a stub in the users main program. Note that even if this code is called by *__main()*, if the WINDOW tool type is not defined, there will be no window.

EXAMPLE

WINDOW=CON:0/0/640/200/Test Window

Technical Information on Floating Point

There are four floating point formats supported by Aztec C68K on the Amiga. Which one you choose depends on whether you are looking for speed, accuracy, or whether you have a math co-processor on your machine.

1. How to create a floating point format program

When using floating point, here are two rules that MUST be followed in order for your program to work correctly.

1. Compile with the desired floating point option.

2. Link with the corresponding floating point library and place that BEFORE the *c.lib*.

2. Floating point formats

The compiler will generate code to use the appropriate format based on one of the following options:

+ff — This is Motorola Fast Floating Point (FFP), the default format. Computations are done in single precision (32 bit) floating point. The appropriate floating point math library to link when using this format is *m.lib*. Float or double register variables can be declared and will be mapped onto registers D4-D7 as appropriate.

+fi — This is the IEEE Double Precision Floating Point Emulation. There is a choice between linking with Manx Aztec C's library (*mx.lib*) or Amiga's library (*ma.lib*). The *ma.lib* math library provides access to the *mathieeedoubbas.library*. Up to two register variables may be declared with this format using the register pair D4-D5 and D6-D7.

+f8 — This is the 68881 Floating Point and the math library *m8.lib* must be linked in. Four register variables may be declared using FP4-FP7.

The default startup code opens the MathFFP library since it is in the Kickstart and does not cost anything. The MathIeeeDoubBas library is opened on the first access of a floating point function when using the *ma.lib*.

For example, if *test.c* was going to access Manx Aztec C's version of the IEEE Double Precision Floating Point Emulation, this is how it would be compiled and linked:

```
cc +fi test.c
ln test.o mx.lib c.lib
```

The compiler (*cc*) defaults to using the Manx IEEE format internally since it does not require the MathIeeeDoubBas.library file in this way. Constants are converted to FFP if necessary on output.

Hybrid combinations of IEEE and FFP format are possible by compiling different modules with different options. This gives speed and accuracy choices to the programmer. Conversion between the two is done using two routines in the IEEE library.

Running On A Single Drive System

Creating a Working Disk

As supplied, the main system disk is fairly full under the assumption, that the second drive will contain all the data files. If only one drive is being used, considerable "fat" can be trimmed from the system disk to provide a reasonable amount of space. There are four areas where files can be deleted. Before deleting any files, make a copy of your distribution disks and delete files from the COPY only.

* First is the *c* directory. There are either 2 or 3 editors here, keep only the one you use. Delete the *Search* command and any other commands that you don't believe are necessary. This will vary from individual to individual, but it's amazing how little you actually need in a normal work session. The only thing you should not delete is the *RUN* command which is used by the CLI.

* Next is the *lib* directory. If you are not using floating point, all the math libraries (*m.lib*, *ma.lib*, *mx.lib* and *m8.lib*) can be deleted. Also *s.lib* can be deleted if you are not using any of the Manx screen functions.

* Third is the *devs* directory which contains drivers for different devices. Unless you are working with the speech synthesizer, definitely delete *narrator.device* and any other devices which you will not be using, such as the *printer.device*.

* Lastly, is the *libs* directory. Here again, the *translator.library* is large and unnecessary if not working with speech. The *mathtrans.library* contains the transcendental floating point math routines. And the *mathieeedoubbas.library* contains the IEEE floating point emulation. The *info.library* is used by the *info* entry in the WorkBench menu.

Once the appropriate files have been deleted, make a copy of this disk and use it as a template for making other disks for different projects.

Building Programs That Run From The WorkBench

1. Overview

This section discusses how to build programs that run from the WorkBench.

First off, *you should read section 4, chapter 2 in volume 1 of the ROM Kernel Manual which discusses the WorkBench.*

Next, there are two ways for a program running from the WorkBench to operate. First, it can completely handle all of its own I/O by creating windows, attaching a console device to the window and getting input from the console and/or Intuition. For this case, there isn't much to discuss since EVERYTHING is handled by the programmer.

The second way is for the program to use the standard input and output channels that are assumed by the program to be pre-opened by the startup code. In this case, the program is doing most or all of its operations as though it were running on a simple line-oriented terminal. The trick here is to get a window opened by the startup code. This is accomplished by the *__wb_parse()* routine, which scans the ToolArray types which are part of the Icon used to invoke the tool. It scans the array for a definition of the form:

WINDOW=xxxxx

If found, the routine will attempt to open the specified string and if successful will set *pr_CIS* and *pr_COS* to the window. Then, in *__main(), stdin, stdout*, and *stderr* will be set up for the window as well. If not found, *stdin, stdout* and *stderr* will not be initialized and any output will be thrown away.

Note that this allows for debugging messages to be placed in a program and only displayed when the appropriate ToolType has been set.

The only other note of importance is that when a program is run from the WorkBench, the initial WorkBench startup message is passed as *"argv"* to main with *"argc"* set to zero. However, for compatibility, the global variable *WBenchMsg* is initialized to point to the startup message as well.

2. Step By Step ...

In this section, we will go through the step by step procedure to produce a program which displays the "Hello world!" message from the

WorkBench. Boot the system using the *SYS1* disk and place the *SYS2* disk in the second drive. Type the following commands:

```
cd ram:
copy * hello.c
main()
{
        printf("Hello world!\n");
        getchar();
}
^\
cc hello.c
ln hello.o -lc
```

At this point, run the *hello* program just to make sure it works okay from the CLI. The reason the *getchar()* routine is called is to give us time to see the message when we run from the WorkBench.

Now, type:

```
sys2:system/iconed
```

The *IconEd* program should start and display a screen with some information. Click the **OK** button and in the menu select clear frame to erase the existing picture. Now exercise your artistic freedom and make any old kind of icon.

Once you have your icon designed, select *save* from the menus. A small requester will appear. Change the file name, *name*, to "hello". Then click in the "Save entire image" box and then close the window to exit the program. Now type the following line:

```
loadwb
```

This command will start up the WorkBench, so resize the CLI's window till you can see the disk icons. Click on the *Ram:* icon and you should see your *Hello* icon. Click on the icon **ONCE**. This selects it but does not attempt to execute it. Now from the *WORKBENCH* menu, select *INFO*.

Selecting *INFO* should bring a window up with a number of gadgets and some information about the icon. Towards the bottom of the window is a single line TOOL TYPES string gadget. Click on the ADD button, click in the string gadget and type the following line:

```
WINDOW=CON:0/0/500/30/Hello
```

Then click on the ADD button again and then click on the SAVE button in the bottom lefthand corner of the window.

That's all there is to it! Double click on the *Hello* icon now and the window should appear with the message waiting for a line to be typed. Hitting return will end the program with the window disappearing.

Creating Sub-Tasks

The fact that the Amiga supports multi-tasking gives the programmer a very powerful tool. Often it is easier to have a separate task handling asynchronous events without tying up the main program. An excellent example of this is the *beep* program in the examples directory. A secondary task is created to actually make the beep sound whenever it gets a message from the parent task.

Creating the sub-tasks is discussed in the Rom Kernel Manual to some extent. However, there are a few special considerations when using Aztec C. In particular, when using small code and data there is a little extra setup that must be done.

When running with small code and data, the A4 register is used as a base register to access global variables. Thus for the program to operate correctly, A4 must point at the proper memory location.

When a task is started, the registers it starts with contain somewhat arbitrary values. It's almost certain that A4 will not contain the appropriate value. Thus, the very first thing the task should do is set up the A4 register. This is done by calling the routine *geta4()* as the first thing in the new task's function.

For example:

```
main()
{
        void subtask();

        CreateTask("TASK", (long)PRI, subtask, (long)STKSIZ);
        .....
}

void subtask()
{
        geta4();
        .....
}
```

Interrupt Service Routines

Interrupt routines are used to provide real-time response to events. Interrupts are serviced by the Exec system routines and if an application program has so specified, an interrupt service routine can be called in response to a particular interrupt.

Now, when the interrupt occurs, the processor can be just about anywhere, with almost anything in its registers. The system interrupt handler will preserve a number of registers and then call the user service routine. However, it will not save all of the registers. In particular, it will not save registers D2, D3 and A4.

If the interrupt service routine modifies these registers, it must first preserve them. Since Aztec C considers D2 and D3 to be temporary registers it does not preserve them. In addition, register A4 is used as a base register to the global data of the program and should be modified if small code and data are being used.

There are three ways of preserving the necessary registers. First, if the program is compiled with the *+p* option, the compiler will preserve D2 and D3 at the entry to every function. In addition, this option forces the large code and data model so register A4 need not be modified and thus need not be saved. Note that the object modules need to be linked with the *lcl32* library.

The second way is to save the registers using in-line assembly language. In this case, the registers need to be restored as well. The code that should be used at the beginning of the routine should be:

```
#asm
     movem.l  d2/d3/a4,-(sp)
     jsr      __geta4#
#endasm
```

Note that we called the *geta4()* function to set up the A4 register. A bit more on that shortly.

The code at the end of the routine should be:

```
#asm
    movem.l  (sp)+,d2/d3/a4
#endasm
```

The third and final way is to use two routines in the library which perform the exact same functions. They are called *int_start()* and *int_end()*. These can be used to achieve the same effect.

As an example, the interrupt handler *intfunc* could look like this:

```
intfunc()
{
   int_start();
      ...
   int_end();
}
```

The reason the A4 register may sometimes need to be saved involves the different memory reference models allowed by Aztec C. In the large model, all references are absolute and 32 bits. In the small model, references are 16 bits and either pc-relative or relative to some base register. For Aztec C, the base register is A4.

Thus, all references are made relative to register A4. When the interrupt occurs, register A4 might be pointing anywhere since almost any task might be executing, not only ours. If A4 is not set up correctly by the interrupt service routine as the very first thing, the interrupt routine might use the A4 register to access some global variables causing who knows what kind of trouble.

Segmentation

1. Segmentation Overview

Segmentation is the approach used by Aztec C to allow a program to successfully execute in a machine with limited memory. It is similar to and replaces the technique generally referred to as *overlays.*

In segmentation, a program can generally be designed with the idea that there are a number of major mutually exclusive functions performed by the program. Thus, it is not necessary for every function to be available at all times. Instead, the part of the program which acts as the control section calls each function when it is needed. The control section always remains in memory and is known as the root.

When the program is segmented, each function is placed in a different segment. When the program is executed, only the root segment is loaded into memory. Then, when the root calls a subroutine which is located in the segment of a specific function, that segment is automatically loaded into memory from disk. The segment remains in memory until the root segment explicitly frees the memory by calling a special library function with the name of a subroutine in the segment to be unloaded.

In reality, any reference from one segment to another segment will cause the specified segment to be loaded. This means that there is really no restriction imposed upon the structure of the program. In fact, one way to understand segmentation is to think of it as simply delayed program loading. If no segment is ever freed, it is possible to load all segments into memory at once if sufficient memory is present.

1.1 Loading a Segment

When a segment is loaded, the call to the segment loader informs it what segment is desired. The segment number is used to index into the segment data table and get the file offset of the segment in the output file. The segment loader seeks to the offset and calls the DOS loader to load the segment. The hunks in the segment are added to the segment list of the program.

Once the segment has been loaded, the segment loader walks through the segment *Jmptab* changing each entry to an absolute reference. It uses the hunk data tables to find the correct load address and the number of entries per hunk.

1.2 Unloading a Segment

When a segment is unloaded, the process occurs in reverse. The absolute jumps are converted back to the *bsr* and *offset*, the hunks of the segment are removed from the segment list and the segment is unloaded.

Segments are only unloaded by manual calls from the program to unload a segment, using the *freeseg()* call. The argument is the address of a function in the segment.

For example, if *foo()* is declared to be a function in the segment to be unloaded, the call to *freeseg()* would look like:

```
int foo();

freeseg(foo);
```

1.3 Segload

The segment loader is a routine which has the name *.segload*. This name is generated by the linker and must be defined in the program. A default version is supplied in the library as well as in the *lib* directory in the file *segload.o*.

There is also a user callable routine, *segload()* which takes the same argument as *freeseg()*, but loads the appropriate segment into memory without actually calling a function in the segment. This can be used by a program which wishes a number of segments to be loaded at the beginning of the program before any other activity occurs.

1.4 Of linking and startup code

To place the library in the root segment after all the other segments have been linked, use:

```
+o0 -lc
```

The startup code for the program must also be in the root. It can be forced in with the library, or can be explicitly included. It is in the library and also in the *lib* directory called *crt0.o* and *lcrt0.o* respectively.

2. Segment Construction

2.1 Segment References

All references between segments are made through the segment *Jmptab* located in the data hunk. Initially, the reference contains the following 8 bytes:

```
bsr     .segload        ;segment loader routine
dc.b    segnum          ;number of segment to load
dc.b    hioffset        ;high 8 bits of 24 bit offset
dc.w    looffset        ;low 16 bits of 24 bit offset
```

After the segment is loaded, each entry looks like:

```
jmp     absolute        ;relocated address of symbol
dc.w    offset          ;saved offset to .segload
```

2.2 Segment Data Table

The segment data table has two parts. The first part consists of eight bytes each of segment data:

```
dc.l    seg_offset      ;offset to segment in file
dc.w    tab_offset      ;offset of segment's entries
dc.w    hunk_offset     ;relative offset of hunk data
```

2.3 Hunk Data

Following the segment data is the hunk data which consists of:

```
dc.w    hunk_num        ;number of the hunk
dc.w    num_symbol      ;number of symbols in Jmptab
```

The hunk data for each segment is terminated by a long word of zero.

Using MANX Technical Support

We have put together a set of guidelines to help you take the most advantage of the technical support service offered by MANX. We ask that you read and follow these guidelines to enable us to continue to give you quality technical support.

Have everything with you.

Try to be organized. When using our phone support, have everything you need with you at the time you call. Our goal is to get you the help you need without keeping you on the phone too long. This can save you a lot of time, and if we can keep the calls as short as possible we can take more calls in the day. This can be to your advantage on days when we are busy and it's hard to get through. Also, *have the following information ready* when you call technical support. We will ask you for this information first.

* *Your name.* This is necessary in case we need to get back to you with additional information.
* *Phone number.* In case we have additional information we will be able to contact you. This will never be given to anyone, so you need not worry.
* The *product* you are using, and the *serial number.* If you have a cross compiler please tell us both host and target, even if the problem is with just one side of the system.
* The *revision of the product* you are using. This should include a letter after the number: i.e. 3.20d or 1.06d. THIS IS VERY IMPORTANT. The full version number may be found on your distribution disks or when you run the COMPILER.
* The *operating system* you are using, and also the version.
* The *type of machine* you are using.
* Anything interesting about your machine configuration. ie. ram disk, hard disk, disk cache software etc.

Know what questions you wish to ask.

If you call with a usage question please try to have your questions narrowed down as much as possible. It is easier and quicker for all to answer a specific question than general ones.

Isolate the code that caused the problem.

If you think you have found a bug in our software, try and create a small program that reproduces the problem. If this program is small enough we will take it over the phone, otherwise we would prefer that you mail it to us, using the supplied problem report, or leave it on one of our bbs systems. Once we receive a "bug report" we will attempt to reproduce the problem and if successful we will try to have it fixed in the next release. If we can not reproduce the problem we will contact you for more information.

Use your C language book and technical manuals first.

We have no qualms about helping you with your general C programming questions, but please check with a C language programming book first. This may answer your question quicker and more thoroughly. Also, if you have questions about machine specific code, i.e. interrupts or dos calls, check with that machine's technical reference manual and/or operating system manual.

When to expect an answer.

A normal turn around time for a question is anywhere from 2 minutes to 24 hours, depending on the nature of the question. A few questions like tracing compiler bugs may take a little longer. If you can call us back the next day, or when the person you talk to in technical support recommends, we will have an in-depth answer for you. But normally we can answer your questions immediately.

Utilize our mail-in service.

It is always easier for us to answer your question if you mail us a letter (We have included copies of our problem report form for your use). This is especially true if you've found a bug with our compiler or other software in our package. If you do mail your question in, try to include all of the above information, and/or a disk with the problem. Again, please write small test programs to reproduce possible bugs. The address for mail-in reports is P.O. Box 55, Shrewsbury, N.J. 07701. If you have questions/problems concerning C Prime or Apprentice C, mail them to P.O. Box 8, Shrewsbury, N.J. 07701.

Updates, Availability, Prices.

If you have any questions about updates, availability of software, or prices, please call our order desk. They can help you better and faster. You can reach them at...

Outside N.J. --> 1-800-221-0440
Inside N.J. --> 1-201-542-2121 (also for outside the U.S.A.)

Bulletin board system.

For users of Aztec C we have a bulletin board system available. The number is ...

1-(201)-542-2793 This is at 300/1200 bps. (all products)

Answer the questions that will be asked after you are connected. When this is done you will be on the system with limited access. To gain a higher access level send mail to SYSOP. Include in this information your serial number and what product you have. Within approximately 24 hours you should have a higher access level, provided the serial number is valid. This will allow you to look at the various information files and upload/download files.

To use the bulletin board best, please do not put large (> 8 lines) source files onto the news system, which we use for an open forum question/answer area. Instead, upload the files to the appropriate area, and post a news item explaining the problem you are having. Also, the smaller the test program, the quicker and easier it is for us to look into the problem, not to mention the savings of phone time.

When you do post a news item, please date it and sign it. This will be very helpful in keeping track of questions. Try to do the same with uploaded source files.

Phone support, number and hours.

Technical support for Aztec C is available between 9:00 am and 6:00 pm eastern standard time at 1-(201)-542-1795. Phone support is available to registered users of Aztec C with the exception of the Apprentice C and C Prime products. For those products, please use the mail-in support service and send questions/problems to P.O. Box 8, Shrewsbury, N.J. 07701.

These guidelines will aid us in helping you quickly through any roadblocks you may find in your development. Thanks for your cooperation.

MANX Problem Report

Date:_____/_____/_____

Name:_____

Phone #:1-(_____)-_____-_____

Company :_____

Address :_____

Product : c86-PC_____ c86-CPM86_____ c68k_____
c68k-Am_____ cII_____ c80_____
c65-ProDos_____ c65-Dos3.3_____
cross:_____

VERSION #:_____ Serial #:_____

Op. - sys.:_____ Machine Config.:_____

Send this form to :

Manx Software Systems
P.O. Box 55
Shrewsbury, N.J. 07701

(C Prime/Apprentice C only):
MANX Software Systems
P.O. Box 8
Shrewsbury, N.J. 07701

or call tech support at 1-201-542-1795 between 9am - 6pm EST.
(Sorry, phone support not available for the C Prime/Apprentice C product.)

Description of problem --
(include what has already been attempted to fix it)
(use the reverse side of this sheet if needed.)

MANX Problem Report

Date:_____/_____/_____

Name:_____

Phone #:1-(_____)-_____-_____

Company :_____

Address :_____

Product : c86-PC_____ c86-CPM86_____ c68k_____
c68k-Am_____ cII_____ c80_____
c65-ProDos_____ c65-Dos3.3_____
cross:_____

VERSION #:_____ Serial #:_____

Op. - sys.:_____ Machine Config.:_____

Send this form to :

Manx Software Systems
P.O. Box 55
Shrewsbury, N.J. 07701

(C Prime/Apprentice C only):
MANX Software Systems
P.O. Box 8
Shrewsbury, N.J. 07701

or call tech support at 1-201-542-1795 between 9am - 6pm EST.
(Sorry, phone support not available for the C Prime/Apprentice C product.)

Description of problem --
(include what has already been attempted to fix it)
(use the reverse side of this sheet if needed.)